MQSeries® Everyplace for Multiplatforms

プログラミング・ガイド

V1.2

SC88-8654-02
(英文原典：SC34-5845-02)

MQSeries® Everyplace for Multiplatforms

プログラミング・ガイド

V1.2

SC88-8654-02
(英文原典：SC34-5845-02)

ご注意

本書の情報およびそれによってサポートされる製品を使用する前に、317ページの『付録C. 特記事項』に記載する一般情報をお読みください。

ライセンスについての警告

MQSeries Everyplace バージョン 1.2 ツールキットにより、開発者は MQSeries Everyplace アプリケーションを作成し、それを実行するための環境を作成することができます。

この製品、または本製品を使用するアプリケーションを実稼働環境に配置する前に、必要なライセンスをもっていることをご確認ください。

指定したサーバー・プラットフォームで MQSeries Everyplace を (コード開発とテストの目的以外で) 使用する場合、各マシンおよびマシン・アップグレードで本プログラムを使用するライセンスを受けるために、capacity-unit 使用許可 (「ライセンス証書」文書に記されていて、発行されている capacity unit table および pricing group table に従って MQSeries Everyplace の使用をサポートするために有効なもの) を入手する必要があります。

指定したクライアント・プラットフォームで本製品を (コード開発およびテストの目的以外で) 使用するには、device platform 使用許可 (「ライセンス証書」文書に記されていて、MQSeries Everyplace の使用をサポートするために有効なもの) が必要です。これらのライセンスは MQSeries Everyplace Bridge を使用したり、IBM によって発行された MQSeries Everyplace pricing group リスト (下記の URL によって Web でも使用可能) に指定されているサーバー・プラットフォームで実行したりする資格をユーザーに与えるものではありません。

これらの制約事項の詳細については、*http://www.ibm.com/software/mqseries* を参照してください。

この版は、MQSeries Everyplace for Multiplatforms V1.2、および新版において特に断りのない限り、それ以降のすべてのリリースとモディフィケーションに適用されます。

本書は、随時改訂され、内容は更新されます。
最新版は、MQSeries ファミリー・ライブラリー Web ページ `http://www.ibm.com/software/ts/mqseries/library/` (英語) にアクセスしてください。

本マニュアルに関するご意見やご感想は、次の URL からお送りください。今後の参考にさせていただきます。

`http://www.ibm.com/jp/manuals/main/mail.html`

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

`http://www.ibm.com/jp/manuals/` の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)

原　典： SC34-5845-02
MQSeries® Everyplace for Multiplatforms
Programming Guide
Version 1.2

発　行： 日本アイ・ビー・エム株式会社

担　当：　ナショナル・ランゲージ・サポート

第1刷 2001.6

この文書では、平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™W7を使用しています。この (書体*) は、(財) 日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、
平成角ゴシック体™W5、平成角ゴシック体™W7

目次

本書について

本書は MQSeries Everyplace for Multiplatforms 製品 (本書では一般に MQSeries Everyplace と呼びます) のプログラミング・ガイドです。本書には、「*MQSeries Everyplace for Multiplatforms プログラミング・リファレンス*」に記述されている MQSeries Everyplace クラス・ライブラリーの使用法に関する説明が記載されています。共通メッセージング・タスクに対してどのクラスを使用したらよいかを判別するのに役立つ情報が載せられており、多くの場合にサンプル・コードも提供されています。

1ページの『第1章 概要』には、MQSeries Everyplace の概念とコンポーネントに精通していない方のために簡単な概要が示されています。 13ページの『第2章 概説』には、環境の設定に役立つ情報があり、またサンプルを使用してアプリケーションを作成する方法が示されています。本書の残りの部分には、MQSeries Everyplace でのプログラミングのさまざまな面についてのより詳細な情報が記載されています。

本書は、「*MQSeries Everyplace for Multiplatforms プログラミング・リファレンス*」および trademark="Java® プログラミングに関する既存の資料またはマニュアルとともに使用することをお勧めします。

本書は、随時改訂され、内容は更新されます。最新版を入手するには、MQSeries ファミリー・ライブラリー Web ページ `http://www.ibm.com/software/mqseries/library/` (英語) にアクセスしてください。

本書の対象読者

本書は、MQSeries Everyplace システム内のセキュア・メッセージ、およびMQSeries Everyplace システムと他の MQSeries ファミリーのメッセージングおよびキューイング製品のメンバーとの間でセキュア・メッセージを交換する Java ベースの MQSeries Everyplace プログラムを作成するプログラマーを対象としています。

Java 以外の開発キットの可用性については、MQSeries Web サイト (`http://www.ibm.com/software/ts/mqseries/`) を参照してください。

前提条件となる知識

本書は、読者に Java およびオブジェクト指向プログラミング技法の実用的な知識があることを前提としています。

セキュア・メッセージングの概念の初歩の知識があると助けになります。そうでないなら、以下の MQSeries 資料をお読みになることをお勧めします。

- *MQSeries An Introduction to Messaging and Queuing*

- *MQSeries Windows NT® 版 インストールの手引き V5.1*、またはユーザーが使用しているオペレーティング・システムに関連する「MQSeries (WinNT/2000) インストール・ガイド」マニュアル。

上記の資料は、オンライン MQSeries ライブラリーの Book セクションからソフトコピーで入手することができます。このライブラリーは、MQSeries Web サイト、 URL アドレス `http://www.ibm.com/software/ts/MQSeries/library/` からアクセスできます。

本書で使用される用語

本書では、以下の用語が使用されています。

MQSeries ファミリー

以下の MQSeries プロダクトを指します。

- **MQSeries Workflow** は、人とアプリケーションを含むビジネス・プロセスを自動化することにより、企業全体での統合を単純化します。
- **MQSeries Integrator** は、リアルタイムでインテリジェントなルールに基づいたメッセージ・ルーティング、コンテンツの変換およびフォーマットを提供する、強力なメッセージ・ブローカー・ソフトウェアです。
- **MQSeries Messaging** は、35 を超えるプラットフォームでサポートされている、ビジネス・クオリティー・メッセージングによりデスクトップからメインフレームへの複数接続を提供します。

MQSeries Messaging

以下のメッセージング・プロダクト・グループを指します。

- **分散メッセージング:** 次のプラットフォームに対応する MQSeries :Windows NT、AIX®、 AS/400®、HP-UX、Sun Solaris、およびその他のプラットフォーム
- **ホスト・メッセージング:** MQSeries for OS/390®
- **ワークステーション・メッセージング:** MQSeries for Windows
- **広範囲メッセージング:** MQSeries Everyplace

MQSeries

以下の 3 つの MQSeries Messaging プロダクト・グループを指します。

- 分散メッセージング
- ホスト・メッセージング
- ワークステーション・メッセージング

MQSeries Everyplace

4 番目の MQSeries Messaging プロダクト・グループである広範囲メッセージングを指します。

デバイス・プラットフォーム
クライアントとしてのみ MQSeries Everyplace を実行することができる小型コンピューター。

サーバー・プラットフォーム
サーバーまたはクライアントとして MQSeries Everyplace を実行することができる、あらゆるサイズのコンピューター。

ゲートウェイ
MQSeries-ブリッジ機能を含む MQSeries Everyplace プログラムを実行する、あらゆるサイズのコンピューター。

変更の要約

このセクションでは、この版の「*MQSeries Everyplace for Multiplatforms プログラミング・ガイド*」に加えられた変更について説明します。本書では、前の版から変更された箇所については、変更箇所の左側に縦線が記されています。

本書 (SC88-8654-02) の変更内容

小さなエラーや脱落が訂正されました。

ハイ・セキュリティー版の説明が除去されました。

次の情報が追加されました。

- ブリッジ・キューのブラウズおよびブリッジ・キューからのメッセージの入手
- 診断ツール
- WebSphere Everyplace Suite (WES) 認証およびプロキシー・サービスの使用
- 新規アダプターの例
- 新規メッセージ保管機能

前の版 (SC88-8654-01) の変更内容

本書の情報の一部を再構成した結果、重複や反復が少なくなりました。小さなエラーや脱落も訂正されました。

次の情報が追加されました。

- AIX および Solaris で MQSeries Everyplace を使用するための詳細情報。
- 読者のためのコメント・フォーム。

変更

第1章 概要

MQSeries Everyplace コードは、パーベイシブ・デバイスやモバイル装置など、広範囲のプラットフォームで実行できます。 MQSeries ホストや分散製品の場合のような、クライアントやサーバーといった考え方はありません。 MQSeries Everyplace キュー・マネージャーは従来のクライアントやサーバーのような働きをもっていますが、各キュー・マネージャーは、実際は、アプリケーション定義タスクを実行できるキュー・マネージャーに過ぎません。

MQSeries Everyplace プログラミング・モデルの基本要素は、メッセージ、キュー、およびキュー・マネージャー です。 MQSeries Everyplace メッセージは、アプリケーション定義コンテンツが含まれているオブジェクトです。保管されたこれらのメッセージはキューで保持され、MQSeries Everyplace ネットワークを介して移動できます。キューは、ローカルであってもリモートであってもよく、キュー・マネージャーによって管理されます。

MQSeries Everyplace キュー・マネージャーは MQSeries Everyplace チャネルを介して通信します。これらのチャネルは要求に応じて作成され、明示的に作成する必要のある MQSeries チャネルと区別するために *動的* チャネルと呼ばれます。これらのチャネルも、*対等* モードと *クライアント / サーバー*・モードの 2 つの方法で構成することができます (7ページの『MQSeries Everyplace チャネル』 を参照)。

MQSeries-ブリッジ コンポーネントも、MQSeries クライアント・チャネルをサポートして、 MQSeries Everyplace ネットワークと MQSeries ネットワークとの通信を可能にします。

2ページの図1 は、MQSeries サーバーにリンクされた MQSeries Everyplace ネットワークの例を示したものです。MQSeries Everyplace オブジェクトとその使用法について、以下の各セクションで概説します。

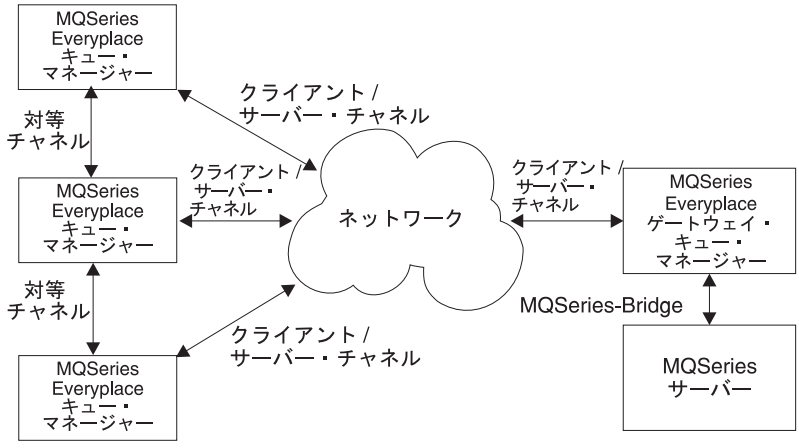

図 1. MQSeries Everyplace クライアント

MQSeries Everyplace キュー・マネージャー

MQSeries Everyplace キュー・マネージャーは、MQSeries Everyplace システムの中心拠点です。これは次のものを提供します。

- MQSeries Everyplace アプリケーションのためのメッセージングおよびキューイング・ネットワークへのアクセスの中央点
- クライアント側キュー (オプション)
- 管理機能 (オプション)
- 1 回だけの確実なメッセージ送達
- 障害状態からの完全なリカバリー
- 拡張可能なルールに基づく動作

MQSeries Everyplace キュー・マネージャーの設計は、オブジェクト指向です。オブジェクトは継承することができ、ルールを使用することによって、キュー・マネージャーの動作をカスタマイズすることができます。 MQSeries Everyplace キュー・マネージャーは、ユーザー作成プログラム内に組み込まれたコードであり、これらのプログラムは MQSeries Everyplace がサポートする任意のデバイスまたはプラットフォームで実行することができます。

キュー・マネージャーはさまざまな 'スタイル' で構成できますが、その主なものは クライアント (対等、またはデバイスとも呼ばれる)、サーバー、およびゲートウェイです。これらのスタイルの詳細については、 57ページの『キュー・マネージャーの開始』を参照してください。

MQSeries Everyplace キュー・マネージャーは、 4ページの『MQSeries Everyplace キュー』で説明されているさまざまなタイプのキューを制御することができます。MQSeries メッセージング・ネットワーク上のほかのキュー・マネージャーとの通信は、同期で行うことも非同期で行うこともできます。同期通信を使用する場合は、発信元と宛先の両方の MQSeries Everyplace キュー・マネージャーがネットワーク上で使用可能になっていなければなりません。非同期通信では、 MQSeries Everyplace アプリケーションは、リモート・キュー・マネージャーがオフラインのときでもメッセージを送信することができます。

MQSeries Everyplace キュー・マネージャーについての詳細は、 45ページの『第4章 キュー・マネージャー、メッセージ、およびキュー』を参照してください。

MQSeries Everyplace キュー

MQSeries Everyplace 環境で使用できるキュー・クラス には、いくつかの異なるタイプがあります。 MQSeries Everyplace 開発パッケージで使用可能なタイプは次のとおりです。

- ローカル
- リモート
- ストア・アンド・フォワード (蓄積交換)
- ホーム・サーバー
- MQSeries-ブリッジ

キューは、認証、圧縮、暗号化などの特性を持つことができます。これらの特性は、属性を使用して設定することができ、メッセージ・オブジェクトがキューに保管されるときに使用されます。

ローカル・キュー

キューの中で最も単純なタイプは、ローカル・キューです。ローカル・キューは、すべてのメッセージの最終的な宛先となる、実際のキューです。このタイプのキューは、特定のキュー・マネージャーに対してローカルに存在し、そのキュー・マネージャーに属します。キュー・マネージャー上のアプリケーションは、そのキュー・マネージャーのキューと直接対話し、安全にメッセージを保管することができます (ハードウェアの障害やデバイスの損失が生じた場合を除く)。これらのキューは、オンラインでもオフラインでも、つまり、ネットワークに接続してもネットワークに接続しなくても使用することができます。

キューはアクセスおよびセキュリティーを所有しており、リモート・キュー・マネージャーは、 (ネットワークに接続する場合) これらの特性を使用することができます。これによって、他のキュー・マネージャーは、そのキューとメッセージをやり取りすることが可能になります。

ローカル・キューについての詳細は、 139ページの『ローカル・キュー』を参照してください。

リモート・キュー

このタイプのキューは、ローカル環境にありません。実際のキューと、そのキューを所有するキュー・マネージャーを識別するための、ローカル・キュー定義があります。

リモート・キューは、同期的にも非同期的にもアクセスできます。リモート・キューのローカル定義がある場合、アクセスのモードはこの定義に基づきます。この場合、アクセスのモードは、同期かまたは非同期かのいずれかになります。ただし、ローカル定義

がない場合、キュー・ディスカバリー が行われます。 MQSeries Everyplace は特性 (認証、暗号、および圧縮) を実際のキューから取得し、アクセスのモードを強制的に同期とします。

リモート・キューについての詳細は、 143ページの『リモート・キュー』を参照してください。

ストア・アンド・フォワード (蓄積交換) キュー

ストア・アンド・フォワード (蓄積交換) キューは、次のキュー・マネージャーが受信できるようになるまでメッセージを保管します。 (これは、キューが属するキュー・マネージャーでない場合もあります。) 通常、このタイプのキューは サーバー上で定義され、クライアントはネットワークに接続する際にそのメッセージを収集します。

ストア・アンド・フォワード (蓄積交換) キューは、多数のクライアントへのメッセージを保持する場合もありますし、クライアントごとに 1 つのストア・アンド・フォワード (蓄積交換) キューが設けられる場合もあります。

ストア・アンド・フォワード (蓄積交換) キューについての詳細は、 147ページの『ストア・アンド・フォワード (蓄積交換) キュー』を参照してください。

ホーム・サーバー・キュー

このタイプのキューは、通常、クライアント上にあり、ホーム・サーバー というサーバー上のストア・アンド・フォワード (蓄積交換) キューに関連付けられています。ホーム・サーバー・キューは、クライアントがネットワーク上で接続するときに、ホーム・サーバーからメッセージを取りだします。

ホーム・サーバー・キューには、通常、ポーリング間隔があり、その間隔で、ネットワークへの接続中にサーバー上に保留メッセージがないかをチェックします。

このキューは、サーバーからメッセージを取り出すと、確実なメッセージ送達を使用してローカル・キュー・マネージャーにそのメッセージを書き込みます。次いで、そのメッセージは宛先キューに保管されます。

ホーム・サーバー・キューについての詳細は、151ページの『ホーム・サーバー・キュー』を参照してください。

MQSeries-ブリッジ・キュー

このタイプのキューは常に MQSeries Everyplace ゲートウェイ キュー・マネージャー上で定義され、 MQSeries Everyplace 環境から MQSeries 環境へのパスを提供します。MQSeries-ブリッジ・キューとは、MQSeries キュー・マネージャーにあるキューを参照する、リモート・キュー定義のことをいいます。

アプリケーションは、ローカル MQSeries Everyplace キューの場合と同じように、このタイプのキューで **put**、**get**、および**ブラウズ**操作を使用できます。

MQSeries-ブリッジ・ブリッジについての詳細は、 153ページの『MQSeries-ブリッジ キュー』を参照してください。

送達不能キュー

MQSeries Everyplace には MQSeries と同様の送達不能キューという概念があります。このキューには送達できなかったメッセージが保管されます。ただし、これらにはその使われ方に重要な違いがあります。

- MQSeries では、メッセージがキュー・マネージャー A からキュー・マネージャー B へ移動しようとしたところ、 A を B に接続するチャネルがそのメッセージを送達できない場合に、メッセージは*受信側*キュー・マネージャーの (B の) 送達不能キューに入れられることがあります。
- MQSeries Everyplaceでは、メッセージがキュー・マネージャー A からキュー・マネージャー B に送信されているものの、実際には送達できなかった場合に、メッセージは*送信側*キュー・マネージャーの (A の) 送達不能キューに入れられます。

  送達不能キューを MQSeries-ブリッジ で使用する場合は特別な注意が必要になります。詳細については、205ページの『配布不能メッセージの処理』を参照してください。

管理キュー

管理キューとは、管理メッセージを処理する方法が収められた特別なキューのことをいいます。

管理キューに書き込まれるメッセージは、内部で処理されます。この理由で、アプリケーションが管理キューから直接メッセージを受け取ることはありません。 1 度に処理されるメッセージは 1 つだけです。 1 つのメッセージが処理されている間に着信する他のメッセージは、キューに入り、着信した順に処理されます。

MQSeries Everyplace チャネル

MQSeries Everyplace は、MQSeries Everyplace チャネル と呼ばれる、キュー・マネージャー間の接続を確立する方式をサポートします。チャネルは、2 つのパーティー間の論理接続であり、データをやり取りする目的で確立されます。

MQSeries Everyplace クライアント および サーバー は、*対等チャネル* および *クライアント / サーバー・チャネル* という 2 つのタイプの接続を介して通信することができます。 MQSeries Everyplace チャネルは要求に応じて作成され、*動的* チャネルと呼ばれます。このチャネルは、明示的に作成する必要のある MQSeries チャネルと区別されます。

クライアント / サーバー・チャネルは、以下のような属性を持っています。

- 要求に応じて作成される。
- チャネル接続は、接続のクライアント側からしか確立できません。
- クライアントは、それぞれの接続が別個のチャネルを使用する多くのサーバーに接続することができます。
- サーバー側キュー・マネージャーは、チャネル・マネージャーやリスナーを使用して、多くの異なるクライアントから多くの接続を同時に受け入れることができます。
- 接続のサーバー側がファイアウォールの背後にある場合は、これらのキュー・マネージャーはファイアウォールを利用します。 (これは、ファイアウォールの構成によって異なります。)
- これらのキュー・マネージャーは *単一方向* であり、同期メッセージングや非同期メッセージングを含め、MQSeries Everyplace によって提供される全範囲の機能をサポートします。

  注: 単一方向とは、クライアントがサーバーへデータを送信したり、サーバーからデータを要求したりできるが、サーバー側はクライアントの要求を開始できないことを意味します。

対等チャネル は、以下のような属性を持っています。

- 要求に応じて作成されます。
- チャネルは、接続のどちらの側からも確立できます。
- キュー・マネージャーは、それぞれの接続が別個のチャネルを使用する多くの他のキュー・マネージャー上の対等チャネル・リスナーに接続することができます。
- キュー・マネージャーは、一時点では 1 つの対等チャネル・リスナーしか持つことができません (現在の制約事項)。つまり、他の 1 つの外部クライアントまたはサーバーしか、任意の一時点でキュー・マネージャーへの対等チャネルを確立できません。この制限は、サーバー・キュー・マネージャーが常に複数の入力要求を並行して処理しようとするため、このチャネル・タイプが、通常、クライアント間でしか使用されないことを意味します。

- 一般に、これらのチャネル・タイプは、ファイアウォールを介して使用するためのものではありません。それは、構成するのが難しいからですが、時には不可能な場合もあります。
- これらのキュー・マネージャーは双方向であり、同期メッセージングや非同期メッセージングを含め、MQSeries Everyplace によって提供される全範囲の機能をサポートします。

  注: 双方向であるということは、チャネルの各端点のキュー・マネージャーがチャネルを介してデータの要求や受け渡しを行うことができることを意味します。

チャネルは、さまざまな属性または特性 (たとえば認証、暗号、圧縮、また使用する伝送プロトコルなど) を持つことができます。さまざまなチャネルがさまざまな特性を使用できます。それぞれのチャネルには、次の各属性ついて独自の値を設定することができます。

認証プログラム
この属性は認証を行えるようにします。これはセキュリティー機能の 1 つであり、アプリケーション環境やユーザーのアイデンティティーを証明するためにものです。

暗号機能
この属性は、チャネルを通過するメッセージについて、暗号化と暗号化解除を行えるようにします。これはセキュリティー機能の 1 つであり、通過中のメッセージをエンコードすることにより、それらをデコードしないで読み取ることができないようにします。

圧縮機能
この属性は、チャネルを通過するメッセージについて、圧縮と解凍を行えるようにします。圧縮機能は、メッセージの送信や保管を行う場合にそのサイズを縮小するためのものです。

宛先
このチャネルの接続先のサーバーおよびポート番号。

認証プログラムは、通常、チャネルをセットアップするときだけ使用され、圧縮機能と暗号機能は、通常、すべてのフローで使用されます。

チャネルについての詳細は、 131ページの『接続』を、そして認証プログラム、圧縮機能、および暗号機能についての詳細は、 221ページの『第8章 セキュリティー』を参照してください。

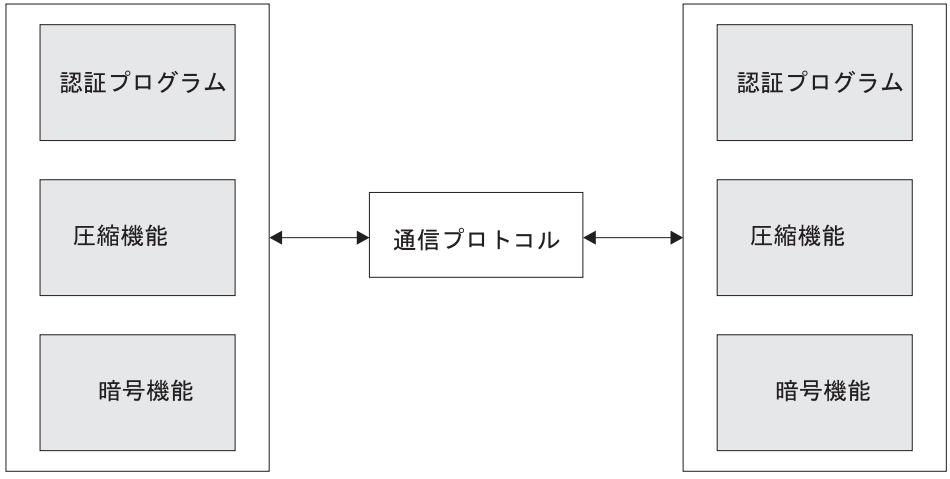

図 2. MQSeries Everyplace チャネル

MQSeries Everyplace チャネルは、各種のプロトコルを使用して確立できますので、多くの方法での接続が可能になります。たとえば、以下の方法が使用できます。

- 永続接続 (LAN、専用回線など)
- ダイヤルアウト接続 (インターネット・サービス・プロバイダー (ISP) に接続する標準モデムの使用など)
- ダイヤルアウトおよび応答接続 (携帯電話、テレビ電話など)

MQSeries Everyplace は通信プロトコルを、アダプターのセットとして、サポートされるプロトコルごとに 1 つずつインプリメントします。このため、新しいプロトコルを非常に簡単に追加することができます。

MQSeries への MQSeries Everyplace ブリッジ

MQSeries Everyplace キュー・マネージャーを、MQSeries サーバーへのインターフェースにすることができます。このタイプのキュー・マネージャーは、ゲートウェイ・キュー・マネージャーと呼ばれます。 MQSeries-ブリッジ は、異なるメッセージ・フォーマット間の変換を含め、 2 つのシステム間のメッセージ転送を処理します。173ページの『MQSeries-ブリッジの構成』で、このインターフェースの詳細が解説されています。

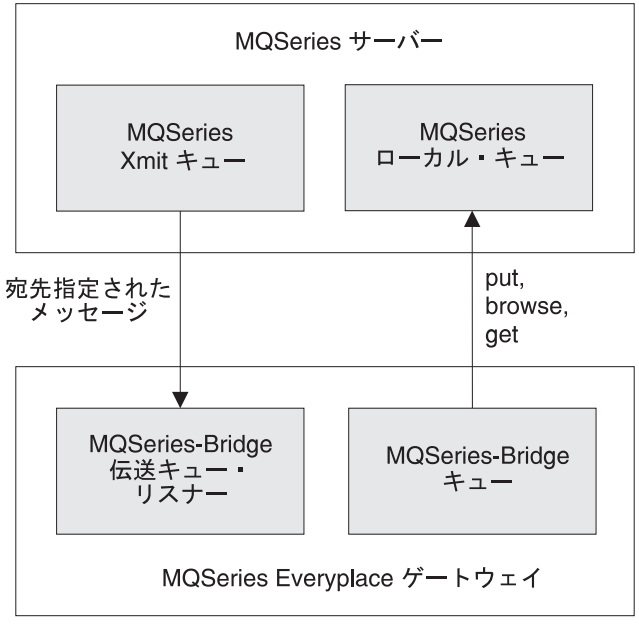

図 3. MQSeries への MQSeries Everyplace インターフェース

セキュリティー

MQSeries Everyplace には、統合された一群のセキュリティー機能が備えられており、メッセージ・データを、ローカルに保管するときも転送するときにも、保護することができます。セキュリティーには、以下の 3 種類のカテゴリーがあります。

ローカル・セキュリティー
ローカル・セキュリティーは、MQSeries Everyplace メッセージがローカル・キュー・マネージャーに保留されている間、それを保護します。

キュー・ベースのセキュリティー
キュー・ベースのセキュリティーは、宛先キューが属性で定義されている限り、開始キュー・マネージャーと宛先キューの間で MQSeries Everyplace メッセージ・データを保護します。この保護は、宛先キューがローカルまたはリモート・キュー・マネージャーのどちらに属しているかには関係ありません。

メッセージ・レベルのセキュリティー
メッセージ・レベルのセキュリティーでは、発信側と受信側の MQSeries Everyplace アプリケーションの間で、メッセージ・データを保護します。

MQSeries Everyplace セキュリティーは、7ページの『MQSeries Everyplace チャネル』に示されている認証プログラム、暗号機能、および圧縮機能を使用します。キュー・ベースのセキュリティーは、 MQSeries Everyplace によって内部処理されますので、メッセージの送信側や受信側が特定のアクションを取る必要はありません。ローカルおよびメッセージ・レベルのセキュリティーは、アプリケーションで開始する必要があります。

MQSeries Everyplace にも、拡張セキュリティーのためのミニ認証サーバーが備えられています。

MQSeries Everyplace セキュリティー機能についての詳細は、 221ページの『第8章 セキュリティー』を参照してください。

注: 世界のいたるところで、暗号のレベルとタイプに関する政府規定が変わりつつあります。必ず、該当する各地の法律規定に従った暗号のレベルとタイプを使用しなければなりません。これは特に、国境を越えて移動するモバイル・デバイスを使用するときに関係があります。MQSeries Everyplace はこのための機能を提供していますが、それをインプリメントするのはアプリケーション・プログラマーの責任です。

第2章 概説

このセクションでは、MQSeries Everyplace 開発キット V1.2 について紹介します。開発キットは、Java 1.1 ベースのメッセージングおよびキュー・アプリケーションを作成するための開発環境です。

注: Java 以外の開発キットの可用性については、MQSeries Web サイト (http://www.ibm.com/software/ts/mqseries/) を参照してください。

開発キットのコード部分は、2 つのセクションに分けられます。

ベース MQSeries Everyplace クラス
メッセージングおよびキュー・アプリケーションを作成するのに必要な機能すべての提供する Java クラスのセット。

例 MQSeries Everyplace の多くの機能を使用する方法を示す Java ソース・コードおよびクラスのセット。

開発環境

MQSeries Everyplace 開発キットを使用して Java でプログラムを開発するには、次のように Java 環境をセットアップしなければなりません。

- Java 開発キット (JDK) が MQSeries Everyplace クラスを見つけられるように *CLASSPATH* を設定します。

  Windows

  たとえば、標準 JDK を使用する trademark=″Windows® 環境では、以下を使用することができます。

  ```
  Set CLASSPATH=<MQeInstallDir>¥Java;%CLASSPATH%
  ```

  UNIX UNIX 環境では、以下を使用することができます。

  ```
  CLASSPATH=<MQeInstallDir>/Java:$CLASSPATH
  export CLASSPATH
  ```

- MQSeries-ブリッジを使用または拡張するコードを開発する場合、 MQSeries Java クラスがインストールされており、 JDK で使用できるようになっていなければなりません。 MQSeries Java クラスの環境を設定することについての詳細は、「*MQSeries Java の使用*」を参照してください。

MQSeries Everyplace と共に数多くの Java 開発環境および Java ランタイム環境を使用することができます。開発とランタイムのどちらのシステム環境も、使用される環境に依存しています。 MQSeries Everyplace には、異なる Java 開発キットについて開発環境をセットアップする方法を示すファイルが含まれています。これは、Windows システムでは、 JavaEnv.bat というバッチ・ファイルで、 UNIX システムでは、 JavaEnv と

いうシェル・スクリプトです。このファイルを使用するには、ファイルのコピーを取って、使用するマシンの環境に適合するようにこのコピーを変更します。

MQSeries Everyplace の例の一部を実行するバッチ・ファイルおよびシェル・スクリプトのセットは、上記の環境ファイルを使用します。このバッチ・ファイルの例を使用するには、環境ファイルを次のように変更しなければなりません。

- *JDK* 環境変数を JDK のベース・ディレクトリーに設定する。
- *JavaCmd* 環境変数を Java アプリケーションを実行するのに使用したコマンドを設定する。
- MQSeries Java クラスがインストールされている場合、 *MQDIR* 環境変数を MQSeries Java クライアントのベース・ディレクトリーに設定する。

注: JavaEnv.bat または JavaEnv のカスタマイズ・バージョンは、 MQSeries Everyplace を再インストールする際に上書きされる場合があります。

JavaEnv.bat を Windows 上で 起動する際、使用する Java 開発キットのタイプを判別するパラメーターを渡さなければなりません。

使用できる値は以下のとおりです。

注: これらのパラメーターには大文字小文字の区別がありますので、示されているとおり正確に入力しなければなりません。

Sun - Sun

JB - Borland JBuilder

MS - Microsoft

IBM - IBM

パラメーターを渡さない場合、デフォルトは IBM です。

UNIX 上の JavaEnv シェル・スクリプトは、対応するパラメーターを使用しません。

Windows では、デフォルトで <MQeInstallDir>\java\demo\Windows ディレクトリーから JavaEnv.bat を実行しなければなりません。UNIX では、デフォルトで、<MQeInstallDir/Java/demo/UNIX ディレクトリーから JavaEnv を実行しなければなりません。このファイルは両方とも、他のディレクトリーから実行したり、他の Java 開発キットを使用したりできるように変更することが可能です。

Windows 2000 および NT セキュリティーの構成

サンプルの Windows NT 認証プログラムが MQSeries Everyplace と一緒に提供されますが、デフォルトの MQSeries Everyplace インストールでは、この認証プログラムの実行に必要なすべての変更が行われるわけではありません。認証プログラムを使用したい場合は、以下の構成を完了する必要があります。

注: Windows NT 認証プログラムは、Supportpac ES02 で出荷された MQe_Explorer によって使用されます。

1. MQSeries Everyplace と Windows セキュリティー間のインターフェースを行う JavaNT.dll ファイルは、検索パスまたは現行ディレクトリーに入っていなければなりません。標準のインストールでは、このファイルは C:\MQe\Java に入っています。Windows .dll ファイルが入っているディレクトリー (通常、 C:\WINNT\system32) にこのファイルのコピーを入れてください。

   注: こうすれば、サンプル認証プログラムがすべての MQSeries Everyplace アプリケーションで使用できるようになります。認証プログラムを MQe_Explorer だけで使用できるようにしたい場合は、 JavaNT.dll のコピーを MQe_Explorer.exe と同じディレクトリーに入れます。

2. Windows ユーザー / パスワード・データベースへのアクセス許可を取得するには、JavaNT.dll に対してセキュリティー許可を正しく設定する必要があります。

   Windows 2000 の場合:

   a. 「**スタート**」ボタンから「**プログラム**」をクリックし、「**管理ツール**」、「**ローカル・セキュリティー・ポリシー**」とクリックします。

   b. 「**ローカル・セキュリティー設定値**」パネルで、左方ペインの「**ローカル・ポリシー**」をクリックし、次に「**ユーザー権利の割り当て**」をクリックします。右方ペインで、現行の *ユーザー ID* に以下のすべての特権が割り当てられていることを確認します。

      - オペレーティング・システムの一部として動作する
      - サービスとしてログオンする
      - ローカルでログオンする

   これらのすべての特権がユーザー ID に割り当てられていない場合は、関係のある特権をダブルクリックし、自分の *ユーザー ID* を追加します。

   Windows NT の場合:

   a. 「**スタート**」ボタンから「**プログラム**」をクリックし、「**管理ツール**」、「**ユーザー・マネージャー**」とクリックします。

   b. 「**ポリシー**」メニューで「**ユーザー権利**」をクリックします。

   c. 「**ユーザー権利ポリシー**」ダイアログで、「**拡張ユーザー権利の表示**」ボックスを調べます。以下の権限を順々に調べます。

      - オペレーティング・システムの一部として動作する

- サービスとしてログオンする
- ローカルでログオンする

それぞれの権限をログオン済みの *ユーザー ID* に与える必要があります。ユーザー ID またはユーザー ID が含まれているグループが、これらのどの権限に対してもリストされていない場合は、「**追加**」ボタンをクリックして自分のユーザー ID を「**権限付与**」リストに追加します。

すべての特権が設定されたら、Windows をログオフしてから、再度ログオンして、それらの特権を現行セッションで使用できるようにします (マシンをリブートする必要はありません)。

アプリケーションの展開

MQSeries Everyplace アプリケーションを展開する場合には、アプリケーションに必要なクラスの最小限のセットを jar 圧縮ファイルに圧縮するようにします。これにより、アプリケーションで必要なシステム・リソースを確実に最小限に抑えられます。 MQSeries Everyplace では、 MQSeries Everyplace クラスが jar ファイルに圧縮される方法について、以下の例を提供しています。これらの例は、標準 MQSeries Everyplace インストールの <MQeInstallDir>¥Java¥Jars ディレクトリーにあります。

MQeDevice.jar
デバイスで使用できるベース・クラスのフル・セット。

MQeGateway.jar
サーバー・プラットフォームで使用できるベース・クラスのフル・セット。

MQeMQBridge.jar
MQeGateway.jar を拡張して、 MQSeries と相互操作するサーバーを作成するのに使用できるクラス。

MQeHighSecurity.jar
MQeGateway.jar および MQeDevice.jar の両方を拡張して、拡張セキュリティーを提供するのに使用できるクラスのセット。

MQeMiniCertificateServer.jar
ミニ認証サーバーを実行するのに必要なすべてのクラスを提供する自己完結型の jar ファイル。

MQeExamples.jar
1 つの jar ファイルへのすべての MQSeries Everyplace 例のパッケージ

MQSeries Everyplace アプリケーションを実行するには、Java ランタイム環境が設定し、必要な MQSeries Everyplace およびアプリケーション・クラスを組み込む必要があります。標準 Java ランタイム環境 (JRE) を使用して、 CLASSPATH に必要な `jar` を組み込むように設定しなければなりません。

以下は、その例です。

Windows

```
Set CLASSPATH=<MQeInstallDir>¥Jars¥MQeDevice.jar;%CLASSPATH%
```

UNIX

```
CLASSPATH=<MQeInstallDir>/Java/Jars/MQeDevice.jar:$CLASSPATH
export CLASSPATH
```

インストール後のテスト

MQSeries Everyplace をインストールしたら、以下の手順で例のセットを実行して、開発キットが正常にインストールされたかどうかを判別します。

- Java 環境が、13ページの『開発環境』で説明されているとおりにセットアップされていることを確認します。このセクションで説明するバッチ・ファイルを実行する場合、それぞれの最初のパラメーターは、使用する Java 開発キットの名前です。名前を指定しない場合、デフォルトは IBM です。

  注: UNIX シェル・スクリプトには、対応するパラメーターがありません。

- 以下の正しいディレクトリーへ移動します。

  Windows
  <MQeInstallDir>¥Java ディレクトリーに変更します。

  UNIX <MQeInstallDir>/Java/demo/UNIX ディレクトリーに変更します。

- キュー・マネージャーを以下のように作成します。

  Windows
  バッチ・ファイルを実行します。

  ```
  CreateExampleQM.bat <JDK>
  ```

  UNIX シェル・スクリプトを実行します。

  ```
  CreateExampleQM
  ```

  上記のようにして、 `ExampleQM` と呼ばれるキュー・マネージャーの例を作成します。

  作成プロセスの一部として、キュー・マネージャー構成情報およびキューを保持するようにディレクトリーをセットアップする作業があります。例では、現行のディレクトリーに対応する ExampleQM というディレクトリーを使用します。このディレクトリーには、さらに 2 つのディレクトリーがあります。

  – Registry - キュー・マネージャー構成データを含むファイルを保持します。
  – Queues - 各キューについて、キューのメッセージを保持するためのサブディレクトリーがあります。 (キューが活動化されるまで、ディレクトリーは作成されません。)

- 以下のように、単純なアプリケーションを実行します。

キュー・マネージャーを作成したら、それを開始し、アプリケーションで使用できます。バッチ・ファイル ExamplesMQeClientTest.bat、またはシェル・スクリプト ExamplesMQeClientTest を使用して、いくつかの単純なアプリケーションの例を実行することができます。

バッチ・ファイルは、デフォルトで、 examples.application.Example1 を実行します。この例では、テスト・メッセージをキュー・マネージャー ExampleQM に書き込んでから、この同じキュー・マネージャーからメッセージを取り出します。この 2 つのメッセージが適合すると、アプリケーションは正常に実行したことになります。

examples.application パッケージにはアプリケーションのセットがあり、MQSeries Everyplace のあらゆる機能をデモンストレーションします。これらの例は、以下のようにして実行できます。

Windows
パラメーターをバッチ・ファイルに渡します。

```
ExamplesMQeClientTest <JDK> <ExampleNo>
```

UNIX パラメーターをシェル・スクリプトに渡します。

```
ExamplesMQeClientTest <ExampleNo>
```

ここで、*ExampleNo* は、例の接尾部です。この範囲は、1 ～ 6 になります。

- キュー・マネージャーを削除します。

  キュー・マネージャーが必要でなくなったら、削除してもかまいません。キュー・マネージャーの例 ExampleQM を削除するには、次のようにします。

  Windows
  バッチ・ファイルを実行します。

  ```
  DeleteExampleQM.bat <JDK>
  ```

  UNIX シェル・スクリプトを実行します。

  ```
  DeleteExampleQM
  ```

  一度キュー・マネージャーを削除すると、それを開始することはできなくなります。

  注:

  1. キュー・マネージャーを削除しても、キューに残っているメッセージや、ベース・キュー・マネージャーの作成の一環として作成されたわけではない構成データは削除されません。したがって、同じ作成パラメーターを使用してキュー・マネージャーが再作成されると、残っているメッセージは再作成されたキュー・マネージャーで使用できることになります。
  2. この例では、セットアップを簡単にするために相対ディレクトリーを使用します。基本の開発およびデモンストレーションの場合以外は、絶対ディレクトリーを使用することを強くお勧めします。現行ディレクトリーを変更した場合、相対ディレクトリーを使用していると、キュー・マネージャー構成情報とキューを見つけることができなくなります。

例

上に説明した例は、MQSeries Everyplace で提供される例のセットの一部に過ぎません。個々の例は、MQSeries Everyplace の機能の使用および拡張方法を説明しています。これらの大半は、ガイドの関係するセクションに説明されています。以下のセクションでは、これらすべてをリストし、簡単に説明します。

examples.adapters

このパッケージには、MQSeries Everyplace アダプター仕様に準拠した 2 つのサンプル・クラスが含まれています。

MQeDiskFieldsAdapter

このサンプル・クラスは、com.ibm.mqe.adapters に含まれているディスク・フィールド・アダプターと機能的に同じものです。このサンプル・クラスは、ローカル・ファイル・ストアでのデータの読み取りや書き込みをサポートします。

WESAuthenticationGUIAdapter

com.ibm.mqe.adapters の中に入っている WESAuthenticationAdapter のラッパーです。この例は、Websphere Everyplace プロキシーの接続時にユーザーにログイン情報を入力するよう促すダイアログ・ボックスを表示することにより、WESAuthenticationAdapter を拡張します。詳細については、 302ページの『Websphere Everyplace Suite (WES) 通信アダプター』を参照してください。

MQSeries Everyplace のアダプターの詳細については、289ページの『第10章 MQSeries Everyplace アダプター』を参照してください。

examples.administration.commandline パッケージ

このパッケージには、コマンド行から基本の MQSeries Everyplace オブジェクトを作成するための一組のサンプル・ツールが含まれています。各プログラムは、管理メッセージの送信方法と応答の解釈方法に関する簡単な例示です。

これらのツールとスクリプトを使用すれば、多くのマシン上でまったく同じ構成を信頼性をもってセットアップすることができます。

これらのツールとそれらを使用する方法の詳細については、163ページの『コマンド行からの管理』を参照してください。

examples.administration.console パッケージ

このパッケージには、MQSeries Everyplace リソースを管理するための簡単なグラフィカル・ユーザー・インターフェース (GUI) をインプリメントするクラスのセットが含まれています。

Admin 管理 GUI の例へのフロントエンド。

加えて、個々の MQSeries Everyplace 管理対象リソースにグラフィカル・ユーザー・インターフェースを提供するクラスのセットがあります。

GUI は、以下のいずれかの方法で呼び出すことができます。

- バッチ・ファイル ExamplesAdminConsole.bat を使用する
- コマンド行から以下を入力する

  ```
  java examples.administration.console.Admin
  ```

- サンプル・サーバーの `examples.awt.AwtMQeServer` のボタンを使用する

MQSeries Everyplace 管理機能の使用についての詳細は、 119ページの『第6章 メッセージング・リソースの管理』を参照してください。

examples.administration.simple パッケージ

このパッケージには、プログラムから MQSeries Everyplace の管理機能の一部を使用する方法を示す例のセットが含まれています。アプリケーションの例と同じく、これらの例も、ローカルまたはリモート・キュー・マネージャーで作業できます。

例 1 キューを作成および削除する

例 2 リモート・キュー・マネージャーへ接続定義を追加する

例 3 キュー・マネージャーの特性を照会し、これ自体をキューに入れる

ExampleAdminBase
すべての管理の例が継承するベース・クラス。

MQSeries Everyplace 管理機能の詳細については、 119ページの『第6章 メッセージング・リソースの管理』を参照してください。

examples.application パッケージ

このパッケージには、キュー・マネージャーと対話するためのあらゆる方法を示す例のセットが含まれています。これには、キューへのメッセージの書き込みおよびキューからのメッセージの取得などが含まれます。すべての例は、ローカル・キュー・マネージャーまたはリモート・キュー・マネージャーのいずれかとともに使用できます。これらのアプリケーションのいずれかを使用できるようにするには、使用するキュー・マネージャーを作成しなければなりません。 Windows 上では CreateExampleQM.bat バッチ・ファイル、UNIX 上では CreateExampleQM シェル・スクリプトを使用して、キュー・マネージャー `ExampleQM` を作成することができます (17ページの『インストール後のテスト』を参照)。

例 1 メッセージの単純な書き込みおよび取得。

例 2 いくつかのメッセージを書き込み、一致するフィールドを使用して 2 番目のメッセージを取得する。

例 3 メッセージ・リスナーを使用して、新しいメッセージの到着を検出する。

例 4 **WaitForMessage** メソッドを使用して、指定された間隔の間に到着するメッセージを取得する。

例 5 メッセージをロック、取得、アンロック、削除する。

例 6 確実なメッセージ送達を使用してメッセージの簡単な書き込みと取得を行う。

例 7 Websphere Everyplace プロキシーを使用してメッセージの簡単な書き込みと取得を行う。

ExampleBase
すべてのアプリケーションの例が継承するベース・クラス。

これらの例は、以下のようして実行されます。

Windows
バッチ・ファイル ExamplesMQeClientTest.bat を使用します

```
ExamplesMQeClientTest  <JDK> <example no> <remoteQMgrName> <localQMgr ini file>
```

UNIX シェル・スクリプト ExamplesMQeClientTest を使用します

```
ExamplesMQeClientTest  <example no> <remoteQMgrName> <localQMgr ini file>
```

ここで、

<JDK> Java 環境の名前 (詳細は 13ページの『開発環境』を参照)。デフォルトは IBM です。

注: このパラメーターは、UNIX では使用されません。

<example no>
実行する例の数 (例の名前の接尾部)。デフォルトは、1 (Example1)。

<remoteQMgrName>
アプリケーションで作業するキュー・マネージャーの名前。これは、ローカルまたはリモート・キュー・マネージャーの名前にすることができます。リモート・キュー・マネージャーの場合、ローカル・キュー・マネージャーがリモート・キュー・マネージャーと通信する方法を定義する接続を構成しなければなりません。

デフォルトでは、ローカル・キュー・マネージャーが使用されます (ExamplesMQeClient.ini で定義されている)。

<localQMgrIniFile>
ローカル・キュー・マネージャーの始動パラメーターを含む ini ファイル。デフォルトでは、ExamplesMQeClient.ini が使用されます。

キュー・マネージャーと対話するアプリケーションの作成方法についての詳細は、 45ページの『第4章 キュー・マネージャー、メッセージ、およびキュー』を参照してください。

examples.attributes パッケージ

このパッケージには、追加のコンポーネントを作成して、MQSeries Everyplace セキュリティーを拡張する方法を示すクラスのセットが含まれています。

NTAuthenticator

ユーザーを Windows NT セキュリティー・データベースに認証する認証プログラム。 NT 認証プログラムは、 Java ネイティブ・インターフェース (JNI) を使用して Windows NT セキュリティーと相互作用します。このコードは、examples.nativecode ディレクトリーにあります。

UnixAuthenticator

UNIX パスワードまたはシャドー・パスワード・システムを使用するユーザーを認証する認証プログラム。 UNIX 認証プログラムは、 JNI を使用してホスト・システムと対話します。このコードは、examples.nativecode ディレクトリーにあります。システムがシャドー・パスワード・ファイルをサポートする場合、 USE_SHADOW プリプロセッサー・フラグを定義して、このネイティブ・コードを再コンパイルしなければなりません。また、コードに対し、実行時にシャドー・パスワード・ファイルを読み取るための十分の権限を与える必要があります。この例は、システムが分散ログオン・サービス (LDAP など) を使用している場合は機能しません。

LogonAuthenticator

基本ログオン認証サポート。

UseridAuthenticator

基本ユーザー ID 認証に対するサポート。

この例では、UserIDS.txt ファイルが入力として必要です。このファイルは、次のようなフォーマットになっていなければなりません。

```
  [UserIDs]
User1Name=User1Password

...

UserNName=UserNPassword
```

TableCryptor

非常に単純な暗号機能

MQSeries Everyplace セキュリティー機能についての詳細は、 221ページの『第8章 セキュリティー』を参照してください。

examples.awt パッケージ

このパッケージは、小さなグラフィカル・インターフェースを必要とするアプリケーションを構築するためのツールキットを提供します。また、MQSeries Everyplace 機能へのグラフィカル・フロントエンドを提供するアプリケーションの例も含まれています。

AwtMQeServer

examples.queuemanager.MQeServer の例へのグラフィカル・フロントエンド。MQeTraceResourceGUI クラスが、 GUI で使用できる国際化されたストリングを含むリソース・バンドルを提供します。 MQeTraceResourceGUI は、パッケージ examples.trace にあります。

バッチ・ファイル ExamplesAwtMQeServer.bat を使用して、このアプリケーションを実行することができます。

サーバー環境でキュー・マネージャーを実行することについての詳細は、64ページの『サーバー・キュー・マネージャー』を参照してください。

AwtMQeTrace

examples.trace.MQeTrace へのグラフィカル・フロントエンド。

MQSeries Everyplace トレース機能についての詳細は、 279ページの『第9章 MQSeries Everyplace でのトレース』を参照してください。

クラス **AwtDialog**、**AwtEvent**、**AwtFormat**、**AwtFrame**、および **AwtOutputStream** は、小さなフットプリントを持った Awt ベースのグラフィカル・アプリケーションを作成するためのツールキットを提供します。これらのクラスは、多くのグラフィカル MQSeries Everyplace 例によって使用されます。

examples.certificates パッケージ

このパッケージには、これらの例に関する詳細情報を入手するためにミニ認証を管理する例 (262ページの『ミニ認証発行サービス』を参照) とミニ認証の使用に関する例が含まれています。

ListWTLSCertificates

この例は、クラス com.ibm.mqe.attributes.MQeListCertificates のメソッドを使用して、レジストリー内のミニ認証をリストするコマンド行プログラムをさまざまな詳細度でインプリメントします。

RenewWTLSCertificates

この例は、クラス com.ibm.mqe.registry.MQePrivateRegistryConfigure のメソッドを使用して、レジストリー内のミニ認証を更新するコマンド行プログラムをインプリメントします。このメソッドは、専用レジストリーでのみ使用する必要があります。

examples.eventlog パッケージ

このパッケージには、異なる機能にイベントのログを記録する方法を示す例がいくつか含まれています。

LogToDiskFile
ディスク・ファイルにイベントを書き込む。

LogToNTEventLog
Windows NT イベント・ログにイベントを書き込む。このクラスは、 JNI を使用して、 Windows NT イベント・ログと対話します。このコードは、examples.nativecode ディレクトリーにあります。

LogToUnixEventLog
イベントを UNIX イベント・ログ (通常は、 /var/adm/messages) に書き込む。このクラスは、 JNI を使用して、 UNIX イベント・ログ・システムと対話します。このコードは、examples.nativecode ディレクトリーにあります。システム上の SYSLOG デーモンを、適切なイベントを報告するように構成する必要があります。

examples.install パッケージ

このパッケージには、キュー・マネージャーを作成および削除するためのクラスのセットが含まれています。

DefineQueueManager
ユーザーがキュー・マネージャーの作成時にオプションを選択できるようにするための GUI。オプションが選択されると、この例は、キュー・マネージャー始動パラメーターを含む ini ファイルを作成して、その後キュー・マネージャーを作成します。

CreateQueueManager
キュー・マネージャーの始動パラメーターの入った ini ファイルの名前およびディレクトリーを要求する GUI プログラム。名前およびディレクトリーが提供されると、キュー・マネージャーが作成されます。

SimpleCreateQM
キュー・マネージャーの始動パラメーターの入った ini ファイルの名前となるパラメーターを使用するコマンド行プログラム。また、オプションで、キューが保管されるルート・ディレクトリーをパラメーターとして指定できます。有効な ini ファイルが見つかると、キュー・マネージャーが作成されます。

DeleteQueueManager
キュー・マネージャーの始動パラメーターの入った ini ファイルの名前を使用する GUI プログラム。有効な ini ファイルが見つかると、キュー・マネージャーは削除されます。

SimpledDeleteQM
キュー・マネージャーの始動パラメーターの入った ini ファイルの名前となるパラメーターを使用するコマンド行プログラム。有効な ini ファイルが見つかると、キュー・マネージャーは削除されます。

GetCredentials
キュー・マネージャーの始動パラメーターの入った ini ファイルの名前を使用する GUI プログラム。有効な ini ファイルが見つかると、キュー・マネージャーのための新しい信任状 (専用 / 共通鍵ペアおよび共通証明書) が取得されます。この処理を正常に行うためには、ミニ認証サーバー が実行されていて、かつ新しい証明書に対する要求が許可されていなければなりません (262ページの『ミニ認証発行サービス』を参照してください)。

すべての構成ファイルは、**ConfigResource** および **ConfigUtils** によって提供されているリソースとユーティリティーを使用します。

キュー・マネージャーの作成および削除についての詳細は、 45ページの『第4章 キュー・マネージャー、メッセージ、およびキュー』を参照してください。

examples.messagestore パッケージ

このパッケージに含まれている例は、 MQeAbstractMessageStore クラスを使用してメッセージ・ストアをインプリメントする方法を示しています。

MessageStore
ローカル・キューなどで使用されるサンプル・メッセージ・ストア。

IndexEntry
この例は、メッセージ索引入力機能を実行します。このクラスの作業は以下のとおりです。

- メッセージ・ストア内のメッセージに対して状態モデルをインプリメントする
- メッセージの保管メカニズムを提供する

クラスはサブクラスに分割できます (たとえば、メッセージの保管に使用するメカニズムを変更するために)。

このクラスは、IndexEntryConstants インターフェースで提供される定数を使用します。

examples.mqbridge.awt パッケージ

このパッケージには、MQSeries-ブリッジ の使用方法および拡張方法を示すクラスのセットが含まれています。例の一部は、他の MQSeries Everyplace の例を拡張します。

AwtMQBridgeServer クラス

これは、基礎 examples.mqbridge.queuemanager.MQBridgeServer クラスへのグラフィカル・インターフェースの例です。

MQBridgeServer クラス・ソース・コードは、以下の指針に従いつつ、ブリッジ機能を MQSeries Everyplace サーバー・プログラムに追加する方法の例を示します。

ブリッジ使用可能サーバーを開始するには

1. 基本 MQSeries Everyplace キュー・マネージャーをインスタンス化して、実行を開始します。
2. 基本 MQSeries Everyplace キュー・マネージャーに渡したのと同じ .ini ファイルを渡して、 com.ibm.mqe.mqbridge.MQeMQBridges オブジェクトをインスタンス化し、その **activate()** メソッドを使用します。

こうしてブリッジ機能は使用可能になります。

ブリッジ使用可能サーバーを停止するには

1. **MQeMQBridges.close()** メソッドを呼び出して、ブリッジ機能を使用不可にします。このようにすることにより、現行の MQSeries-ブリッジ操作すべてを完全に停止し、すべての MQSeries-ブリッジ機能をシャットダウンします。
2. MQeMQBridges オブジェクトをガーベッジ・コレクションされたものとして、このオブジェクトへの参照を除去します。
3. 基本 MQSeries Everyplace キュー・マネージャーを停止およびクローズします。

ExamplesAwtMQBridgeServer.bat

このファイルでは Awt サーバーを使用して MQBridgeServer を呼び出す方法の例が提供されています。これには、 AwtMQBridgeTrace モジュールの初期設定を制御する方法も示されています。

ExamplesAwtMQBridgeServer.ini

このファイルは、MQSeries-ブリッジ機能をサポートするキュー・マネージャー用の構成ファイルの例です。

MQSeries-ブリッジについての詳細は、 173ページの『第7章 MQSeries-ブリッジ』を参照してください。

examples.mqbridge.administration.commandline パッケージ

このパッケージには、一組のサンプル・ツールが含まれています。これらのサンプル・ツールは、 examples.administration.commandline パッケージに含まれているものと類似していて、 MQSeries-ブリッジ に必要なオブジェクトを管理するためのものです。

これらのツールとそれらを使用する方法の詳細については、163ページの『コマンド行からの管理』を参照してください。

examples.nativecode パッケージ

いくつかの例は、 Windows NT または UNIX (AIX または Solaris) 上のオペレーティング・システム機能へのアクセスを必要とします。 MQSeries Everyplace は、 JNI を使用してこれらの機能にアクセスします。 Windows の場合、 examples¥native ディレクトリーにあるコードは、 `examples.attributes.NTAuthenticator` および `examples.eventlog.LogToNTEventLog` によって必要とされる JNI インプリメンテーションを提供します。 UNIX の場合、ファイル examples/native/JavaUnix.c のコードは、 `examples.attributes.UnixAuthenticator` および `examples.eventlog.LogToUnixEventLog` によって必要とされる JNI インプリメンテーションを提供します。

examples.queuemanager パッケージ

キュー・マネージャーは、数多くの異なるタイプの環境で実行できます。このパッケージには、キュー・マネージャーが、クライアント、サーバー、またはサーブレットとして実行できるようにする例のセットが含まれています。

MQeClient
通常はデバイスで使用される単純なクライアント

MQePrivateClient
セキュア・キューおよびセキュア・メッセージングで使用できるクライアント。

MQeServer
複数の複数のキュー・マネージャー (クライアントまたはサーバー) に同時に接続できるサーバー。通常、サーバー・プラットフォームで使用されます。バッチ・ファイル ExamplesAwtMQeServer.bat を使用すると、このサーバーにグラフィカル・フロントエンドを提供する `examples.awt.AwtMQeServer` の例を実行できます。

MQePrivateServer
MQeServer と類似しているものの、セキュア・キューおよびセキュア・メッセージングを可能にする。

MQeServlet
サーブレットでキュー・マネージャーを実行する方法を示す例。

MQeChannelTimer
チャネル・マネージャーをポーリングして、アイドル・チャネルをタイムアウトにできるようにする例。

MQeQueueManagerUtils
各種の MQSeries Everyplace コンポーネントの構成を開始するヘルパー・メソッドの集合。

異なる環境でキュー・マネージャーを実行することについての詳細は、 57ページの『キュー・マネージャーの開始』を参照してください。セキュア・キューおよびメッセージングの環境を提供するキュー・マネージャー (MQePrivateClient および MQePrivateServer) についての詳細は、 221ページの『第8章 セキュリティー』を参照してください。

examples.rules パッケージ

ベース MQSeries Everyplace 機能は、ルールを使用して制御および拡張できます。MQSeries Everyplace のコンポーネントの一部は、ルール・クラスがこれらに適用できるようにします。これらのルールは、コンポーネントの機能を変更する手段を提供します。このパッケージには、以下のルール・クラスの例が含まれています。

ExamplesQueueManagerRules
保留メッセージを伝送するよう定期的に試行するキュー・マネージャー・ルール・クラスの例。

詳細については、107ページの『第5章 ルール』を参照してください。

AttributeRule
属性の使用を制御する属性ルールの例。

examples.security パッケージ

このパッケージには、MQSeries Everyplace セキュリティーを変更する例が含まれています。

MQeSecurity
MQSeries Everyplace の特定の機能の使用を許可するかどうかを制御する Java セキュリティー・マネージャーへの拡張機能の例。

examples.trace パッケージ

このパッケージには、開発時のアプリケーションのデバッグ、および完了したアプリケーションのトレースに使用できるトレース・ハンドラーの例が含まれています。

MQeTrace
ベース MQSeries Everyplace トレース・クラス。

AwtMQeTrace。 examples.awt パッケージにあり、 MQeTrace クラスへのグラフィカル・フロントエンドを提供します。

MQeTraceResource

MQSeries Everyplace により出力されるトレース・メッセージを含むリソース・バンドル

MQeTraceResourceGUI

このクラスには、トレース・ウィンドウ制御用のすべての変換可能テキストが含まれます。

例

第3章 MQeFields

MQeFields は、MQSeries Everyplace メッセージを送信、受信、または操作するためにデータ項目を保持するのに使用される基本クラスです。 MQeFields オブジェクトは次のように構成されています。

```
/* create an MQeFields  object */
MQeFields fields = new MQeFields( );
```

MQeFields 内には、アイテムを格納および検索するためのさまざまな **put** および **get** メソッドがあります。アイテムは名前、型、および値の形式で保持されます。

名前は以下のルールに従っている必要があります。

- 1 文字以上である。
- ASCII 文字セットに準拠していなければならない (20 < value < 128 の値の文字)。
- { } [ ] # ( ) : ; , ' " = といった文字を含めてはならない。
- MQeFields オブジェクト内で固有でなければならない。

MQeFields オブジェクト名は値の検索および更新に使用されます。MQeFields オブジェクトがダンプされるときに名前はデータとともに組み込まれるので、名前は短くしておくのがよいでしょう。

次の例は、MQeFields オブジェクトに値を格納する方法を示しています。

```
/* Store integer values into a fields object */
  fields.putInt( "Int1", 1234 );
  fields.putInt( "Int2", 5678 );
  fields.putInt( "Int3",    0 );
```

次の例は、MQeFields オブジェクトから値を検索する方法を示しています。

```
/* Retrieve an integer value from a fields object */
  int Int2 = fields.getInt( "Int2" );
```

表1に示されている値の型を格納および検索するためのメソッドが提供されています。

表 1. 格納および検索メソッド

値の型	格納メソッド	検索メソッド
byte (バイト)	putByte	getByte
int (整数)	putInt	getInt
short (短精度)	putShort	getShort
long (長精度)	putLong	getLong
floating point (浮動小数点)	putFloat	getFloat
	putDouble	getDouble

表 1. 格納および検索メソッド (続き)

値の型	格納メソッド	検索メソッド
boolean (ブール)	putBoolean	getBoolean
string (ストリング)	putAscii	getAscii
	putUnicode	getUnicode

値の配列はフィールド・オブジェクト内に保持できます。配列の保持には 2 つの形式があります。

- 固定長配列は、**putArrayOf***type*、および **getArrayOf***type* メソッドを使用して処理されます。 *type* は、 Byte、Short、Int、Long、Float、または Double になります。
- 可変長配列は、**put***type* **Array** および **get***type* **Array** を使用して処理されます。 *type* は、 Byte、Short、Int、Long、Float、または Double になります。

  この形式を使用することにより、各エレメントは単一のアイテムを連続させた形で格納されます。アイテムの名前に *:nn* が付加されます。ここで、*nn* は 0 で始まる配列内のアイテムのエレメント番号です。個々のアイテムに配列長が含まれます。この配列長は整数値で、**putArrayLength**、および **getArrayLength** を使用して処理されます。

MQeFields オブジェクトは、 **putFields** および **getFields** メソッドを使用して別の MQeFields オブジェクト内に組み込むことができます。

MQeMsgObject、またはこのクラスの下位クラスは、通常の MQSeries Everyplace メッセージで使用されます。 MQeMsgObject は MQeFields クラスの下位クラスなので、すべての MQeFields メソッドにアクセスできます。 MQeMsgObject について詳しくは、 83 ページの『メッセージ』を参照してください。

MQeFields オブジェクトの内容は、以下の形式でダンプされます。

バイナリー

これは、ネットワークを介して MQeFields または MQeMsgObject オブジェクトを送信する際に通常使用される形式です。データをバイナリーに変換するために用いられるメソッドは **dump** です。このメソッドは、オブジェクトの内容をエンコードした形式を含むバイナリー・バイト配列を戻します。 **dump** メソッドにはオプションとしてブール値パラメーターがあり、これはダンプされたデータを直前のオブジェクト・データのコピーで XOR 処理するかどうかを指定します。これは、出力配列の "0x00" であるバイト数を増やして、ネットワークを介して送信されるデータ・ストリームのサイズを圧縮機能が小さくできるようにするという方法です。このパラメーターは、アプリケーションが他の物理メディアにバイト配列を書き出すことを意図している場合にのみ役立ちます。

固定長配列がダンプされ、配列にエレメントが全く含まれない (長さがゼロの) 場合、その値はヌルとして復元されます。

エンコード・ストリング

ストリング形式にはさまざまな制限が課せられており、ストリングを使用して MQeFields オブジェクトを格納することは必ずしも可能ではありません。ストリング形式は、 MQeFields オブジェクトの **dumpToString** メソッドを使用します。これには 2 パラメーター、テンプレートと名称が必要です。テンプレートは、次の例が示すように、 MQeFields 項目データをどのように返還する必要があるかを示すパターン・ストリングです。

```
"(#0)#1=#2¥r¥n"
```

この場合

#0 データ・タイプ (たとえば、ascii または short)

#1 フィールド名

#2 値のストリング表記

その他の文字は変更されずに出力ストリングにコピーされます。メソッドは組み込み MQeFields オブジェクトをオブジェクトに正常にダンプしますが、組み込み MQeFields データが **restoreFromString** メソッドを使用して復元できることの保証はありません。

MQeFields の強力な機能として、次の例が示すように、 ini ファイルを読み込む、というものがあります。

```
[Section1]
Keyword1=value1
Keyword2=value2
[Section2]
Keyword1=value
...
```

次の例が示すように、このデータは読み込みおよび構文解析して MQeFields オブジェクトに取り込むことができます。

```
File diskFile = new File( fileName );              /*access the file*/
  byte data[] = new byte[(int) diskFile.length()];       /*file size*/
  FileInputStream inputFile = new FileInputStream( diskFile );
  inputFile.read( data );                             /*read the file*/
  inputFile.close( );                               /*finish with file*/
  MQeFields fields = new MQeFields( );          /*new Fields Object*/
  fields.restoreFromString( "¥r¥n",            /*end of line string*/
"[#0]",                                          /*section pattern*/
"#1=#2",                                          /*keyword pattern*/
byteToAscii( data ) );
```

次の例のように、コードの変形を使用すると、異なるデータ・タイプを復元することができます。

```
[Section1]
(ascii)Keyword1=value1
(int)Keyword2=1234
[Section2]
(boolean)Keyword1=true
...
File diskFile = new File( fileName );               /*access the file*/
  byte data[] = new byte[(int) diskFile.length()];     /*size of file*/
  FileInputStream inputFile = new FileInputStream( diskFile );
  inputFile.read( data );                             /*read the file*/
  inputFile.close( );                               /*finish with file*/

  MQeFields fields = new MQeFields( );            /*new Fields Object*/
  fields.restoreFromString( "¥r¥n",              /*end of line string*/
"[#0]",                                             /*section pattern*/
"(#0)#1=#2",                                        /*keyword pattern*/
byteToAscii( data ) );
```

注: dumpToString メソッドは、前述の技法を使用して復元できない形式には、組み込み MQeFields オブジェクトをダンプしません。

ini ファイルの使用は任意ですが、使用することをお勧めします。 ini ファイルを処理するためのユーティリティーは、 MQSeries Everyplace とともに提供されている例にありますが、独自に作成したり、 MQeFields オブジェクトを直接使用することもできます。 ini ファイルを使用する場合、キュー・マネージャーを作成および管理するために、ファイルが MQeFields オブジェクトに回復されなければなりません。この処理のために役立つ例が提供されています。

MQeFields ベースの ini ファイル・エディターの作成

この例は、 MQSeries Everyplace examples.awt ディレクトリーにあるコンポーネントの例を使用して、 ini ファイル・エディターを作成します。これはすべての形式の MQeFields を含むという意味はなく、より強力なエディターの例または出発点を意図したものです。

次の例は、各セクションを別個の組み込み MQeFields オブジェクトとして扱います。基本クラスは、ini ファイル内にあるすべてのセクションをリストする選択ボックスの付いたウィンドウを作成します。

この例では、examples.awt ディレクトリーにあるクラスを利用します。これらのクラスにより、基本フレームとダイアログを簡単に作成および操作することができます。

アプリケーションは `examples.awt.AwtFrame` を拡張して、メニューのついたフレームを作成します。

```
public class Editor extends examples.awt.AwtFrame
  /*-----------------------------------------------------------------*/
  public editor( String args[] ) throws Exception
    {
  /* Assign the title to the frame and initializes the ancestor.*/
    super( "Editor - " );
  /*Assign a menu bar to the frame and define the items that appear on the bar*/
    format( Menu,   new String[][][] {
                          { { "File"      },
                            {  " ", "Open"  },       /* Index   0   */
                            {  " ", "Save"  },       /* Index   1   */
                            {  "-" },
                            {  " ", "Exit"  } },     /* Index   2   */
                            { { "Help"    },
                            {  " ", "Trace" } } } ); /* Index   3   */
    visible( true );
    }
```

format メソッド呼び出しには 2 つのパラメーターがあります。この例における最初のものは *Menu*、 2 番目のものが *String array* オブジェクトです。

String array 次のように、3D 配列でなければなりません。

- 最初のディメンションが行数を定義する。
- 2 番目のディメンションが列数を定義する。
- 3 番目のディメンションがメニューのコンポーネントを定義する。これらは次のように定義されます。

  ```
  new String[][][] { type, Data {, Data, { ... } } }
  ```

  この場合

  type コンポーネントのタイプ。これは以下のいずれかになります。

″ ″	標準のメニュー項目
″**C**″	CheckItem - チェックなし。修飾子 ″!″ は、チェックされたことを示します。
″**-**″	separator

  他のものはすべてラベルとして扱われる。

  Data コンポーネントによって使用されるテキスト。

  アクション・イベントを引き起こしうるメニューの各項目には、先行するコード断片の配列での位置に基づいてインデックス番号が付けられています。コメントにインデックス番号が示されています。

format メソッドの最初のパラメーターには、 "North"、"South"、"East"、 "West"、および "Center" という値を割り当てることができます。これらは、フレーム内のパネルの位置に対応します。この場合、 *string array* オブジェクトは次のような構文になります。

```
new String[][][] { type, Data {, Data, { ... } } }
```

この場合

type コンポーネントのタイプ。これは以下のいずれかになります。

- ″**A**″ テキスト域。次の修飾子を指定することが可能です。
  - ″**P**″ 保護 - 編集不能
  - ″**K**″ キー解放アクション・イベントを与える
- ″**B**″ ボタン
- ″**C**″ チェック・ボックス - チェックなし。修飾子 ″!″ は、チェックされたことを示します。
- ″**D**″ 選択 (ドロップダウン・リスト)
- ″**L**″ ラベル
- ″**S**″ 選択リスト (リスト・ボックス)
- ″**T**″ テキスト・フィールド。次の修飾子を指定することが可能です。
  - ″**K**″ キー解放アクション・イベントを与える
  - ″**P**″ 保護 - 編集不能
  - ″*****″ マスクされた入力

他のものはすべてラベルとして扱われる

Data コンポーネントによって使用されるテキスト

これらのメソッドがどのように働くかについての詳細は、 examples.awt ディレクトリーの AwtDialog、AwtFormat、および AwtFrame の例を参照してください。

エディターの作成に examples.awt コンポーネントを使用することにより、次のコードは 3 つの作業変数、およびメニューと Choice ボックスが 1 つ付いたウィンドウを作成するコンストラクターを定義します。

```
public class Editor extends examples.awt.AwtFrame
  {
  protected Choice    choiceBox   = null;
  protected MQeFields fields      = null;
  protected String    currentFile = "";

  /*------------------------------------------------------------------*/
  public editor( String args[] ) throws Exception
```

```
  {
  super( "Editor - " );
  format( Menu,   new String[][][] {
                        { { "File"      },
                          {  " ", "Open"  },       /* Index   0   */
                          {  " ", "Save"  },       /* Index   1   */
                          {  "-" },
                          {  " ", "Exit"  } },     /* Index   2   */
                          { { "Help"    },
                          {  " ", "Trace" } } } ); /* Index   3   */
  format( North,  new String[][][] {
                        { { "D", "< -- No File Loaded -- >" } } } );
  choiceBox = (Choice) getObject( North,  0 );
  visible( true );
  }

/*--------------------------------------------------------------------*/
public static void main( String[] args )
  {
  try
    {
    new  Editor( args );
    }
  catch ( Exception e )
    {
    e.printStackTrace();
    }
  }
```

次のコードは、ユーザーがメニューと選択ボックスで対話することによって引き起こされたイベントを処理します。

メニュー・アクションは次のとおりです。

開く (Open)
: Action インデックスは "0"。これは switch ステートメントで使用され、 **Load** メソッドを呼び出してディスク・ファイルを読み取る

保管 (Save)
: Action インデックスは "1"。これは switch ステートメントで使用され、 **Save** メソッドを呼び出してディスク・ファイルに書き出す

終了 (Exit)
: Action インデックスは "2"。これは switch ステートメントで使用され、プログラムを終了する

トレース (Trace)
: Action インデックスは "3"。これは switch ステートメントで使用され、Examples.Awt.AwtMQeTrace クラスを呼び出す

選択ボックスは North にだけ配置されているコンポーネントで、そのためインデックス "0" が付けられています。このリスト・ボックスから項目を選択すると、 MQeFields オブジェクトの内容を表示するために使用されるクラスがアクティブになります。

```
public void action( Object e, int where, int index,
   String choice, boolean state )
    {
    try
      {
      switch ( where )
        {
        /* process Menu actions                                          */
        case Menu:
          switch ( index )
            {
            case  0: load( );          break;
            case  1: save( );          break;
            case  2: System.exit( 0 ); break;
            case  3: new examples.awt.AwtMQeTrace( "Edit Trace", null );
    break;
            }
          break;
        /* process North events                                           */
        case North:
          switch ( index )
            {
            case 0:
              String item = choiceBox.getSelectedItem();
              new EditorFieldsDisplay( "Editor - ["  + Item +  "]",
                                       fields.getFields( Item ) );
              break;
            }
              break;
        }
      }
    /* exception occured - show error in a modal dialog window         */
    catch ( Exception ex )
      {
      ex.printStackTrace();
      new examples.awt.AwtDialog( this,
   "Exception",
   examples.awt.AwtDialog.Show_OK,
   new String[][][] {
  { { "TP", ex.toString() }  } } );
      }
    }
```

次のコードは、「保管 (Save)」メニュー要求を処理します。

このコードは、共通ファイル・ダイアログを作成しおよび表示し、出力ファイル名を指定できるようにします。一度ファイル・パスとファイル名が設定されると、すべての

MQeFields オブジェクトを含む String が一つ作成され、組み込まれた各 MQeFields オブジェクトが String 変数にダンプされます。この String はディスクに書き込まれ、出力ファイルはクローズされます。

```
protected void save( ) throws Exception
    {
    if ( fields == null )   throw new Exception( "No Fields object" );
    FileDialog fd  = new FileDialog( this, "", FileDialog.SAVE );
    fd.setFile( CurrentFile );
    fd.show( );
    if ( (fd.getDirectory() != null) && (fd.getFile() != null) )
      {
      currentFile   = fd.getDirectory() + fd.getFile();
      File diskFile = new File( currentFile );
      /* look for imbedded fields objects   */
      String buffer = "";                /*for imbedded_fields objects*/
      String base   = "";                  /*non-imbedded_fields items*/
      Enumeration keys = fields.fields();              /*get the names*/
      while ( keys.hasMoreElements() )
        {
        String key = (String) keys.nextElement();
        if  ( fields.dataType( key )  == MQeField.TypeFields )
          buffer = buffer + "[" + key + "]¥r¥n" +
                   fields.getFields( key ).dumpToString(
   "(#0)#1=#2¥r¥n" ) + "¥r¥n";
        else                                      /*... no, normal item*/
          base   = base   + fields.dumpToString( "(#0)#1=#2¥r¥n", key );
        }
      buffer = base + buffer;
      FileOutputStream outputFile = new FileOutputStream( diskFile );
      outputFile.write( MQe.asciiToByte( buffer ) );
      outputFile.close( );
      }
    }
```

ここまでで、以下のことを行うことができました。

- アプリケーションを制御するウィンドウの定義
- ディスク・ファイルのロードおよび保管の処理の定義
- 特定のセクションが選択された場合に EditorFieldsDisplay クラスをアクティブにするメカニズムの定義

EditorFieldsDisplay クラスは実際の編集が行われる場所です。このクラスはエディター画面を作成します。クラスのコンストラクターは以下をセットアップします。

- メニュー (この場合は North に終了 (Exit) のみ)
- 組み込み MQeFields オブジェクトの名前をすべて含む選択ボックス
- 中央にダンプされた項目を保持するリスト・ボックス

次のコードは画面上にサブウィンドウを配置します。

```
public class EditorFieldsDisplay extends AwtFrame
  {
  protected MQeFields fields     = null;        /* fields object       */
  protected Choice    choiceBox  = null;        /* Fields Choice       */
  protected List      listBox    = null;        /* listbox object      */
  protected String    newItem    =
  "  <<<< Double click here to add new item >>>>";

  /* constructor                                                       */
  public EditorFieldsDisplay( String    thisTitle,
                              MQeFields theseFields ) throws Exception
    {
    super( thisTitle );
    fields = theseFields;
    format( Menu,   new String[][][] {
                              { { "Exit"    },
                                {  " ", "Exit"   } } } );/* Index  0   */
    format( North,  new String[][][] {
                              { { "D",  "<none>" } } } );/* Index  0   */
    format( Center, new String[][][] {
                              { { "S",  "" } } } );      /* Index  0   */

    choiceBox = (Choice) getObject( North,  0 );
    listBox   = (List)   getObject( Center, 0 );
    listBox.setFont( new Font( "Courier", 1, 12) );
    visible( true );
    /* re-size/re-position the edit window                             */
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    setSize    ( screenSize.width / 3, screenSize.height / 3 );
    setLocation( screenSize.width / 3, screenSize.height / 3 );
    /* initialise the various component contents                       */
    showFields( ); /* show fields contents  */
    }
```

showFields メソッド呼び出しは、リスト・ボックスのデータを最新表示する共通ルーチンの呼び出しです。これは、 MQeFields オブジェクト内に保持される項目のリストです。

```
protected void showFields( ) throws Exception
    {
    listBox.removeAll();              /* clear all entries           */
    if ( fields != null )             /* fields object ?             */
      {                               /* ... yes                     */
      Enumeration keys = fields.Fields();       /* get the key names   */
      choiceBox.removeAll();
      while ( keys.hasMoreElements() )
        {
        String key = (String) keys.nextElement();
        if  ( fields.dataType( key )  == MQeField.TypeFields )
          choiceBox.add( key );       /* ... yes, add name           */
        else
          listBox.add( format( fields.dumpToString( "#1¥t(#0)¥t = #2",
         Key), 10 ) );
```

```
      }
    listBox.add( newItem );          /* add information line         */
    }
  }
```

直前のコードに書かれている ListBox.add(format(fields.dumpToString("#1¥t(#0)¥t = #2" レコードは、タブ ("¥t")、復帰 ("¥r")、および改行 ("¥n") 文字で MQeFields データをダンプします。これらはリスト・ボックスに表示される前にフォーマットすることが必要です。

次のコードが示しているのはフォーマッターです。

```
public static String format( String data, int tabSize )
    {
    int l = 0;                                        /*start line number*/
    char c[] = new char[data.length()];                    /*work array*/
    data.getChars( 0, data.length(), c, 0 );       /*convert to chars*/
    StringBuffer result = new StringBuffer( 512 );
    for ( int i = 0; i < c.length; i = i + 1 )
      switch ( c[i] )
        {
        case '¥r': /* ignore */
        case '¥n': /* new line */
           l = 0; /* set space count */
           result.append( c[i] );                    /*append to string*/
           break;
        case '¥t':                                      /*tab character*/
          int m = l;                                  /*current position*/
          for ( l = m; l < tabSize + 1; l = l + 1 )        /*fill tab*/
            result.append( ' ' );                            /*pad*/
          l = 0;                                           /*reset*/
          break;
        default:                                          /*all others*/
          result.append( c[i] );                     /*append to string*/
          l = (l + 1) % tabSize;                  /*increase the length*/
            break;
        }
    return( result.toString() );
    }
```

次のコードは、ユーザーがメニュー、選択ボックスまたはリスト・ボックスで対話することによって引き起こされたイベントを処理します。

メニュー項目は「終了 (Exit)」だけで、これは編集ウィンドウを閉じることによって処理されます。選択ボックスは組み込み MQeFields 項目をすべて処理します。個々の項目を編集するには、ユーザーはリスト・ボックス内の項目を選択する必要があります。

```
  public void action( Object e, int where, int index,
   String choice, boolean state )
    {
    try
      {
      switch ( where )
```

```
    {
    /* process Menu events                                          */
    case Menu:
      switch ( index )
        {
        case  0:  dispose( );       break;
        }
      break;
    /* process North events                                         */
    case North:
      switch ( index )
        {
        case  0:         break;
        }
      break;
    /* process Center events                                       */
    case Center:
      switch ( index )
        {
        case 0:
          int i = listBox.getSelectedIndex();
          if  ( i > -1 )                    /* anything selected ? */
            {
            String editName  = listBox.getItem( i );
            if ( editName.equals( newItem ) )    /* add new item ? */
              editItem( "", "ascii", "" );  /* ... yes,           */
            else
              {
              editName = editName.substring( 0,
                                             editName.indexOf( ' ' ) );
              editItem( editName,
                        fields.dumpToString( "#0", editName ),
                        fields.dumpToString( "#2", editName ) );
              }
            }
          break;
        }
       break;
    }                               /* end switch( Where )          */
  }
/* exception occured - show error in a modal dialog window         */
catch ( Exception ex )
  {
  ex.printStackTrace();
  new AwtDialog( this,
                 "Exception",
                 AwtDialog.Show_OK,
                 new String[][][] { { { "TP", ex.toString() }  } } );
  }
}
```

ユーザーがリスト・ボックスの項目を選択すると、「編集 (Edit)」ダイアログが表示されます。このダイアログで、名前、型、値を編集できます。また、ユーザーは MQeFields オブジェクトから項目を削除することもできます。

同じダイアログを使って、 MQeFields オブジェクトに新しい項目を追加することもできます。この場合、項目名と値はブランクで、型にはデフォルトで ascii が設定されます。

```
protected void editItem( String name, String type, String value )
    throws Exception
    {
    if ( fields == null )   throw new Exception( "No Fields object" );
    /* Dialog to set Field Item name type and value                    */
    AwtDialog md = new AwtDialog( this,
                                  getTitle() + " - edit item",
                                  AwtDialog.Show_OK_Cancel,
                                  new String[][][] {
        { { "L", "Field Item Name:" }, { "T", name  } },    /* Index 1 */
        { { "L", "      Data type:" }, { "D", type,         /* Index 3 */
                                               "ascii",
                                               "boolean",
                                               "byte",
                                               "double",
                                               "float",
                                               "int",
                                               "long",
                                               "short",
                                               "unicode" } },
        { { "L", "          Value:" }, { "T", value    } }, /* Index 5 */
        { { "L", "   Remove item ?" }, { "C", "Delete" } }  /* Index 7 */
                                                   } );
    /* process dialog response                                        */
    if ( md.getActionIndex( South ) == md.Button_OK )
      {
      name = md.GetText( Center, 1 );
      if ( name.equals( "" ) )
        throw new Exception( "Invalid Item name" );
      fields.delete( name );
      if ( ! md.getCheckState( center, 7 ) )   /* delete this item ?   */
        {                                     /* ... no               */
        String data = "(" + md.GetText( Center, 3 ) +
                      ")" + name +
                      "=" + md.getText( Center, 5 );
        fields.restoreFromString( "(#0)#1=#2", data );
        }
      showFields( );
      }
    }
```

これで、実用的ではあるものの素朴な ini ファイル・エディターの完成です。データがエンコードされていない限り、 MQeMsgObjects を表示または修正するのに、このエディターを使用することができます。

第4章 キュー・マネージャー、メッセージ、およびキュー

3ページの『MQSeries Everyplace キュー・マネージャー』では、 MQSeries Everyplace キュー・マネージャーのサービスとキューの概要を示しています。この節では、キュー・マネージャーの機能と使用法、およびそれに関連するリソース、ならびにメッセージとキューについて詳しく説明します。

キュー・マネージャーの作成および削除

キュー・マネージャーは少なくとも、次のものを必要とします。

- レジストリー (60ページの『キュー・マネージャーの MQeRegistry パラメーター』を参照)
- キュー・マネージャー定義
- ローカル・デフォルト・キュー定義 (90ページの『キュー』 を参照)

一度これらの定義が行われると、キュー・マネージャーを実行し、管理インターフェースを使用して、構成 (キューの追加など) をさらに実行することができます。

これらの初期オブジェクトを作成するためのメソッドは、MQeQueueManagerConfigure クラスで提供されます。

インストール・プログラムの例 `examples.install.SimpleCreateQM` および `examples.install.SimpleDeleteQM` は、このクラスを使用します。

このセクションでは、 MQeQueueManagerConfigure クラスの使用について詳しく説明します。

キュー・マネージャーの作成

キュー・マネージャーを作成するために必要な基本ステップは、次のとおりです。

1. MQeQueueManagerConfigure のインスタンスを作成し、アクティブにする
2. キュー・マネージャー特性を設定し、キュー・マネージャー定義を作成する
3. デフォルト・キューの定義を作成する
4. MQeQueueManagerConfigure インスタンスをクローズする

1. MQeQueueManagerConfigure インスタンスの作成および活動化

次のいずれかの方法で、 MQeQueueManagerConfigure クラスをアクティブにすることができます。

1. 空のコンストラクターを呼び出した後、 **activate()** を実行する。

```
try
{
 MQeQueueManagerConfigure qmConfig;
```

```
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 qmConfig = new MQeQueueManagerConfigure( );
 qmConfig.activate( parms, "MsgLog:qmName\\Queues\\" );
}
catch (Exception e)
{ ... }
```

2. 次のようにパラメーターを指定してコンストラクターを呼び出す。

```
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
}
catch (Exception e)
{ ... }
```

最初のパラメーターは、キュー・マネージャーの初期化パラメーターを含む MQeFields オブジェクトです。これらは少なくとも次のものを必要とします。

- 組み込み MQeFields オブジェクト (*Name*)。キュー・マネージャーの名前が入ります。
- 組み込み MQeFields オブジェクト。レジストリー・タイプ (*LocalRegType*) とレジストリー・ディレクトリー名 (*DirName*) の形で、ローカル・キュー・ストアの位置が入ります。基本ファイル・レジストリーを使用する場合、必要なパラメーターはこれだけです。私用レジストリーを使用する場合は、これに加えて *PIN* および *KeyRingPassword* が必要です。

ディレクトリー名は、キュー・マネージャー定義の一部として保管され、それ以降のキュー定義でキュー・ストアのデフォルト値として使用されます。ディレクトリーは、必要になったら作成されるので、存在していなくてもかまいません。

初期化パラメーターのいずれかが別名を使用する場合 (52ページの『別名の使用』を参照)、または別名を使用してチャネル属性ルール名 (47ページの『2.キュー・マネージャー特性の設定およびキュー・マネージャー定義の作成』を参照) を設定する場合には、MQeQueueManagerConfigure をアクティブにする前に別名を定義する必要があります。

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 MQeFields qmgrFields  = new MQeFields();
 MQeFields regFields   = new MQeFields();
```

```
 // Queue manager name is needed
 qmgrFields.putAscii(MQeQueueManager.Name, "qmName");
 // Registry information
 regFields.putAscii(MQeRegistry.LocalRegType, "FileRegistry");
 regFields.putAscii(MQeRegistry.DirName, "qmname¥¥Registry");

 // add the imbedded MQeFields objects
 parms.putFields(MQeQueueManager.QueueManager, qmgrFields);
 parms.putFields(MQeQueueManager.Registry, regFields);
 // set aliases
 MQe.alias("FileRegistry", "com.ibm.mqe.registry.MQeFileSession");
 MQe.alias("ChannelAttrRules", "examples.rules.AttributeRule");
 // activate the configure object
 qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
}
catch (Exception e)
{ ... }
```

2.キュー・マネージャー特性の設定およびキュー・マネージャー定義の作成

MQeQueueManagerConfigure をアクティブにしたら、キュー・マネージャー定義を作成する前に、次のキュー・マネージャー特性の一部またはすべてを定義することができます。

- **setDescription()** を使用すれば、キュー・マネージャーに説明を追加することができます。
- **setChannelTimeout()** を使用すれば、チャネル・タイムアウト値を設定することができます。
- **setChnlAttributeRuleName()** を使用すれば、チャネル属性ルールの名前を設定することができます。

キュー・マネージャー定義を作成するには、 **defineQueueManager( )** を呼び出します。これは、以前に設定された特性を含んでいる、キュー・マネージャー用のレジストリー定義を作成します。

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 // set aliases
 MQe.alias("FileRegistry", "com.ibm.mqe.registry.MQeFileSession");
 MQe.alias("ChannelAttrRules", "examples.rules.AttributeRule");
 // activate the configure object
 qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
 qmConfig.setDescription("a test queue manager");
```

```
 qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
 qmConfig.defineQueueManager();
}
catch (Exception e)
{ ... }
```

この時点で、 MQeQueueManagerConfigure を**クローズ**して、キュー・マネージャーを実行することができます。ただし、キュー・マネージャーにはキューがないので、多くのことは行えません。管理インターフェースを使ってキューを追加することもできません。なぜなら、管理メッセージを保守するための管理キューがないからです。

次のセクションでは、キューの作成方法およびキュー・マネージャーを役立てる方法を説明します。

3.デフォルト・キューの定義の作成

MQeQueueManagerConfigure により、次のような、キュー・マネージャーの 4 つの標準キューを定義することができます。

- 管理キュー : **defineDefaultAdminQueue()**
- 管理応答キュー : **defineDefaultAdminReplyQueue()**
- 送達不能キュー : **defineDefaultDeadLetterQueue()**
- デフォルトのローカル・キュー : **defineDefaultSystemQueue()**

これらのメソッドはすべて、キューがすでに存在する場合には、例外を出します。

管理キューおよび管理応答キューは、キュー・マネージャーが管理メッセージに応答できるようにするため (たとえば、新規の接続定義およびキューを作成するため) に必要です。

正しい送信先に送達できないメッセージを保管するため、送達不能キューを使用することができます (試行されているルールに基づきます)。

デフォルトのローカル・キュー SYSTEM.DEFAULT.LOCAL.QUEUE は、MQSeries Everyplace 自体の中では特別な意味はありませんが、MQSeries Everyplace が MQSeries メッセージ機能とともに使用されるときに役に立ちます。なぜなら、それはすべての MQSeries メッセージング・キュー・マネージャーに存在するからです。

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
 qmConfig.setDescription("a test queue manager");
```

```
 qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
 qmconfig.defineDefaultAdminQueue();
 qmconfig.defineDefaultAdminReplyQueue();
 qmconfig.defineDefaultDeadLetterQueue();
 qmconfig.defineDefaultSystemQueue();
}
catch (Exception e)
{ ... }
```

4. MQeQueueManagerConfigure インスタンスのクローズ

キュー・マネージャーおよび必要なキューを定義したら、 MQeQueueManagerConfigure を**クローズ**して、キュー・マネージャーを実行することができます。

次はその完全な例です。

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 MQeFields qmgrFields  = new MQeFields();
 MQeFields regFields   = new MQeFields();
 // Queue manager name is needed
 qmgrFields.putAscii(MQeQueueManager.Name, "qmName");

// Registry information
 regFields.putAscii(MQeRegistry.LocalRegType, "FileRegistry");
 regFields.putAscii(MQeRegistry.DirName, "qmname¥¥Registry");

 // add the imbedded MQeFields objects
 parms.putFields(MQeQueueManager.QueueManager, qmgrFields);
 parms.putFields(MQeQueueManager.Registry, regFields);

 // set aliases
 MQe.alias("FileRegistry", "com.ibm.mqe.registry.MQeFileSession");
 MQe.alias("ChannelAttrRules", "examples.rules.AttributeRule");

 // activate the configure object
 qmConfig = new MQeQueueManagerConfigure( parms, "MsgLog:qmName\\Queues\\" );
 qmConfig.setDescription("a test queue manager");
 qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
 qmConfig.defineQueueManager();
 qmconfig.defineDefaultAdminQueue();
 qmconfig.defineDefaultAdminReplyQueue();
 qmconfig.defineDefaultDeadLetterQueue();
 qmconfig.defineDefaultSystemQueue();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

直ちにキュー・マネージャーと必要なキューのレジストリー定義が作成されます。キューはアクティブにされるまで作成されません。

キュー・マネージャーの削除

キュー・マネージャーを削除するために必要な基本ステップは、次のとおりです。

1. 管理インターフェースを使用して、定義を削除する。
2. MQeQueueManagerConfigure のインスタンスを作成し、アクティブにする。
3. 標準キューおよびキュー・マネージャー定義を削除する。
4. MQeQueueManagerConfigure インスタンスをクローズする。

これらのステップが完了すると、キュー・マネージャーは削除され、それ以上実行することができなくなります。キュー定義は削除されますが、キューそのものは削除されません。キューに残ったメッセージは、アクセス不能になります。

注: キューにメッセージがある場合、自動的には削除されません。アプリケーション・プログラムには、キュー・マネージャーを削除する前に、残ったメッセージがあるかをチェックしてそれらを処理するためのコードが含まれている必要があります。

1.定義の削除

MQeQueueManagerConfigure を使用すれば、それを使用して作成した標準キューを削除することができます。その他のキューを削除するには、 MQeQueueManagerConfigure を呼び出す前に、管理インターフェースを使用する必要があります。

2. MQeQueueManagerConfigure インスタンスの作成および活動化

このプロセスは、キュー・マネージャーを作成するときと同じです。 45ページの『1. MQeQueueManagerConfigure インスタンスの作成および活動化』を参照してください。

3. 標準キューおよびキュー・マネージャー定義の削除

次のものを呼び出してデフォルト・キューを削除します。

- 管理キューを削除するには、**deleteAdminQueueDefinition()**
- 管理応答キューを削除するには、**deleteAdminReplyQueueDefinition()**
- 送達不能キューを削除するには、**deleteDeadLetterQueueDefinition()**
- デフォルトのローカル・キューを削除するには、**deleteSystemQueueDefinition()**

これらのメソッドは、キューが存在していなくても正常に作動します。

deleteQueueManagerDefinition() を呼び出すことによってキュー・マネージャー定義を削除します。

```
import com.ibm.mqe.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
```

```
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
 qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteAdminQueueDefinition();
 qmConfig.deleteAdminReplyQueueDefinition();
 qmConfig.deleteDeadLetterQueueDefinition();
 qmConfig.deleteSystemQueueDefinition();
 qmConfig.deleteQueueManagerDefinition();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

デフォルト・キューとキュー・マネージャーの定義は、**deleteStandardQMDefinitions()** を呼び出して一緒に削除することができます。このメソッドは便宜上提供されており、次のものと同等です。

```
deleteDeadLetterQueueDefinition();
deleteSystemQueueDefinition();
deleteAdminQueueDefinition();
deleteAdminReplyQueueDefinition();
deleteQueueManagerDefinition();
```

4. MQeQueueManagerConfigure インスタンスのクローズ

キューおよびキュー・マネージャー定義を削除したら、 MQeQueueManagerConfigure インスタンスを**クローズ**できます。

次はその完全な例です。

```
import com.ibm.mqe.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialize the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
 qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteStandardQMDefinitions();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

別名の使用

別名は、クラス名およびその他の MQSeries Everyplace オブジェクトに割り当てることができます。別名は MQSeries Everyplace によって使用されますが、アプリケーションと実際のクラス名との間の間接レベルを提供するためにアプリケーション・プログラムが使用することもあります。そのため、別名が参照するオブジェクト・インスタンスを変更するときに、アプリケーションがそれを行う必要はありません。別名を使用することで、構成を容易に変更することができます。たとえば、キューにはいくつかの別名を与えることができ、これらの任意の名前に送信されたメッセージは、このキューによって受け入れられます。

以下の例は、キューやキュー・マネージャーで使用できる別名を割り当てるいくつかの方法を示しています。

キュー別名割り当ての例

マージ・アプリケーション

以下のような構成であると仮定します。

- データをキュー Q1 に書き込むクライアント・アプリケーション
- Q1 からデータを取り出して処理するサーバー・アプリケーション
- データをキュー Q2 に書き込むクライアント・アプリケーション
- Q2 からデータを取り出して処理するサーバー・アプリケーション

上の 2 つのサーバー・アプリケーションは、後で、両方のクライアント・アプリケーションからの要求をサポートする 1 つのアプリケーションにマージされます。これで、 2 つのキューを 1 つのキューに変更できるようになりました。たとえば、Q2 を削除し、Q1 キューの別名を追加し、それを Q2 と呼ぶことができます。以前 Q2 を使用したクライアント・アプリケーションからのメッセージは、自動的に Q1 に送信されます。

アプリケーションのアップグレード

以下のような構成であると仮定します。

- キュー Q1
- Q1 からメッセージを取得するアプリケーション
- メッセージを Q1 に書き込むアプリケーション

次に、メッセージを取得する新しいアプリケーションを開発します。この新しいアプリケーションを Q2 キューで作業するようにできます。 Q2 キューを定義し、それを使ってこの新しいアプリケーションを実行することができます。このアプリケーションをそのまま使いたい場合は、古いアプリケーションに Q1 キューのすべてのトラフィックを消去させてから、 Q1 という Q2 の別名を作成することができます。 Q1 に書き込むアプリケーションは引き続き機能しますが、メッセージは Q2 からは出なくなります。

単一キューへの異なる転送モードの使用

MQE1 キュー・マネージャーに MY_Q_ASYNC があると仮定します。別のキュー・マネージャー MQE2 が、非同期キューとして定義されたリモート・キュー定義を使用して、メッセージを MY_Q_ASYNC に渡します。ここで、 MY_Q_ASYNC キューから同期的な方法でメッセージを定期的に取得する場合を想定します。

これを行うための推奨方法は、MY_Q_ASYNC キューに、たとえば MY_Q_SYNC という別名を追加することです。次に、 MQE2 キュー・マネージャー上にリモート・リファレンスを定義し、それに MY_Q_SYNC キューを参照させます。こうすることで、 2 つのリモート・キュー定義が得られます。 MY_Q_ASYNC 定義を使用すると、これらのメッセージは非同期的に移送されます。 MY_Q_SYNC 定義を使用すると、同期メッセージ転送が使用されます。

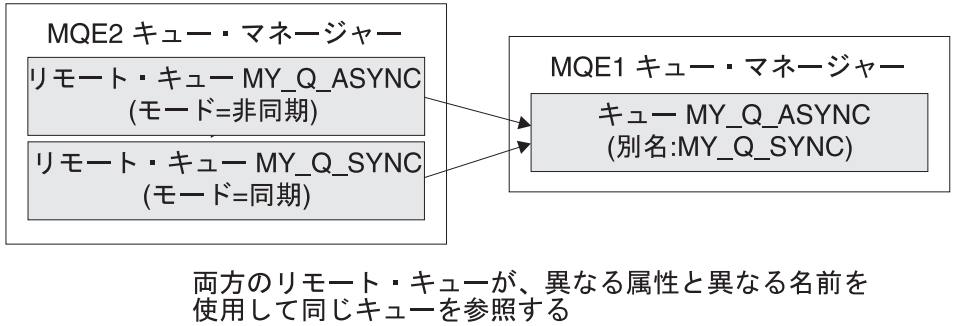

図 4. 単一キューへの 2 つの転送モード

キュー・マネージャー別名割り当ての例

いくつかの異なる名前を使用した場合のキュー・マネージャーのアドレッシング

キュー・マネージャー SERVER23QM がサーバー SAMPLEHOST 上にあり、ポート 8082 を listen していると仮定します。このキュー・マネージャーにアクセスする SERVICEX アプリケーションが、キュー・マネージャーを SERVICEXQM として参照しようとしています。この参照は、以下のようにして、キュー・マネージャーの別名を使用して行うことができます。

- **SERVER23QM での接続の構成:**

  接続名 / 宛先キュー・マネージャー:
  SERVICEXQM

  説明: SERVER23QM が、SERVICEXQM に送信されたメッセージを受信できるようにする別名定義

  チャネル: ヌル

  ネットワーク・アダプター: ヌル

ネットワーク・アダプター・オプション:
ヌル

- **SERVER23QM キュー・マネージャーでのローカル・キューの作成**

  キュー名: SERVICEXQ

  キュー・マネージャー: SERVER23QM

  サーバー・サイド・アプリケーションは、このキューからメッセージを受け取り、それらを処理し、クライアントへ返送します。

これで、サーバー の JVM 内で稼働する MQSeries Everyplace アプリケーションは、 SERVER23QM または SERVICEXQM のいずれかのキュー・マネージャーの SERVICEXQ にメッセージを書き込むことができます。いずれの場合も、メッセージは SERVICEXQ に到着します。

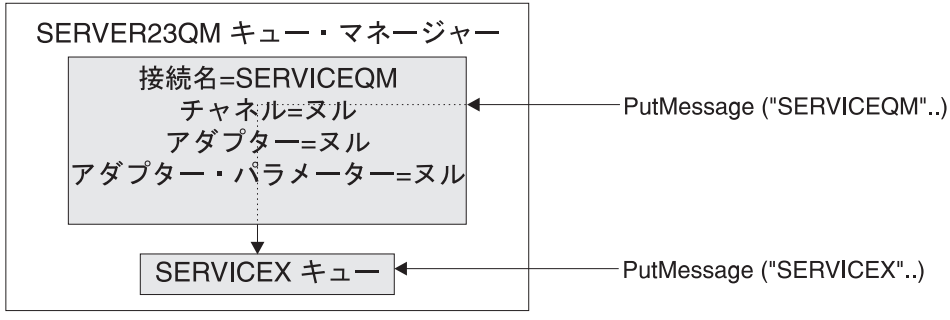

図 5. 異なる 2 つの名前によるキュー・マネージャーのアドレッシング

SERVICEXQ キューを他のキュー・マネージャーへ移動した場合は、接続別名を新規キュー・マネージャーにセットアップできるため、アプリケーションを変更する必要はありません。

キュー・マネージャー間のいくつかの異なる経路指定

上記のシナリオを使用すれば、モバイル装置 (MOBILE0058QM) 上の MQSeries Everyplace キュー・マネージャーは、いくつかの異なる方法で SERVICEXQ キューにアクセスすることができます。 2 つの例を示します。

- **送信側での別名割り当て**

  このメソッドの経路指定を使用した場合、受信側のキュー・マネージャーは、送信側のキュー・マネージャーがそれに別名を割り当てていることを知りません。別名割り当ては、送信側のキュー・マネージャーにのみ限定されています。

  モバイル装置の場合:

– MOBILE0058QM キュー・マネージャーから SERVER23QM キュー・マネージャーへの接続を作成します。

接続名 SERVER23QM

ネットワーク・アダプター・パラメーター

Network:SAMPLEHOST:8082

– SERVER23QM キュー・マネージャーに対して SERVICEXQM という別名を作成します。

あるメッセージがモバイル装置から SERVICEXQM キュー・マネージャーへ送信されると、 MQSeries Everyplace は、SERVICEXQM 名を接続中の SERVER23QM にマップし、そのメッセージを SERVER23QM キュー・マネージャーへ送信します。

そのとき Mobile58QM がそのメッセージを別のサーバー・キュー・マネージャー Server24QM へ送信したい場合は、別名 SERVICEXQM を Server23QM 接続から除去し、それを Server24QM 接続に追加します。こうすれば、受信側のキュー・マネージャーも送信側のアプリケーションも影響を受けることがありません。

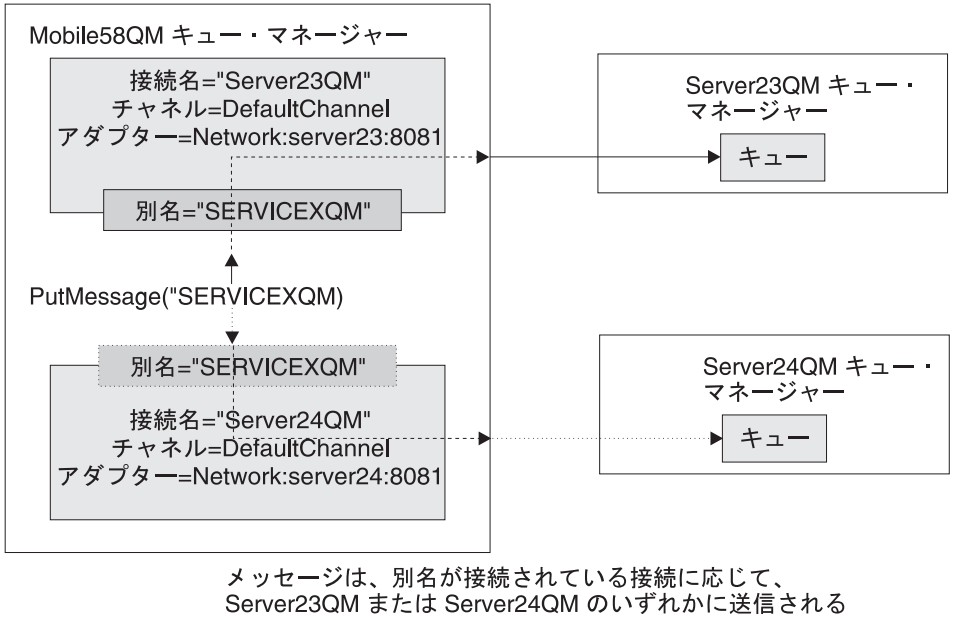

図 6. 異なる 2 つの名前によるキュー・マネージャーのアドレッシング

- **受信側の仮想キュー・マネージャー**

  このメソッドを使用した場合、送信側のキュー・マネージャーは、そのメッセージが中間キュー・マネージャーを経由して宛先キュー・マネージャーに

到着するものと考えます。宛先キュー・マネージャーは、実際には存在しません。「中間」キュー・マネージャーは、この仮想宛先キュー・マネージャーあてのすべてのメッセージ・トラフィックを取り込みます。

モバイル装置の場合:

– MOBILE0058QM キュー・マネージャーから SERVER23QM キュー・マネージャーへの接続を作成します。

  接続名 SERVER23QM

  ネットワーク・アダプター・パラメーター
  Network:SAMPLEHOST:8082

– 最初の接続を介してメッセージを発送する SERVICEXQM への 2 番目の接続を作成します。

  接続名 SERVICEXQM

  ネットワーク・アダプター・パラメーター
  SERVER23QM

  注: これは別名ではありません。これは*経由ルーティング* であり、SERVICEXQM あてに送信されるメッセージが受信側の SERVER23QM キュー・マネージャーを経由して送信されることを示しています。

モバイル装置上の経由ルーティングは、 SERVICEXQM に書き込まれたすべてのメッセージが Server23QM へ送信されるようにします。 Server23QM はそれらのメッセージを取得し、それらが SERVICEXQM キュー・マネージャーあてに送信されるものであることを記憶します。また、SERVICEXQM 名を解決し、それが Server23QM キュー・マネージャー (それ自身) を表す別名であることを認識します。 Server23QM キュー・マネージャーは、次に、それらのメッセージを受け入れ、それらをキューに入れます。

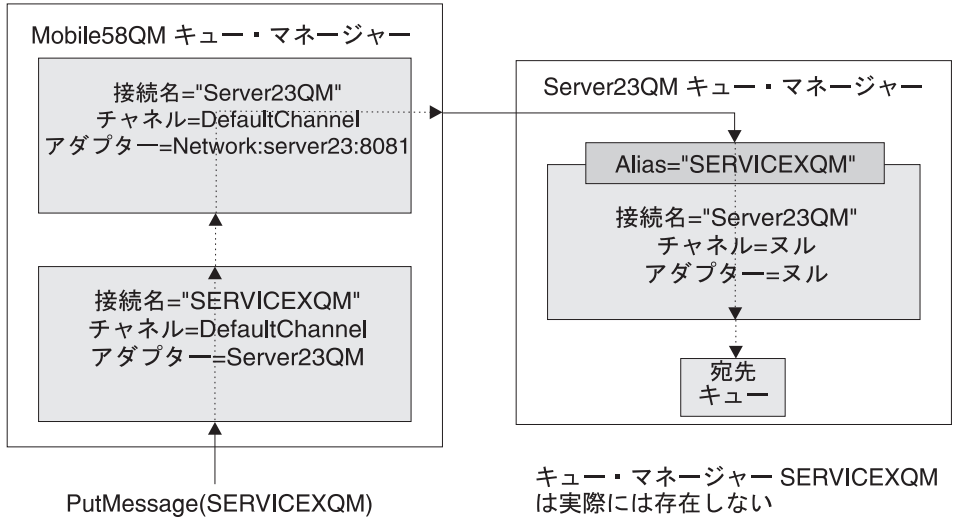

図7. 2 つの名前によるキュー・マネージャーのアドレッシング

上記の操作の代替方法として、SERVICEXQM を存続させることができますが、それを元のマシンから同じ (しかし、JVM が異なる) マシンへ Server23QM キュー・マネージャーとして移動することができます。SERVICEXQM は、別のポートで listen する必要があります。このため、Server23QM から SERVICEXQM への接続も変更する必要があります。

キュー・マネージャーの開始

キュー・マネージャーは、次のように実行することができます。

- クライアントとして
- サーバー内で
- サーブレット内で

次のセクションでは、examples.queuemanager パッケージ内にある、クライアント、サーバー、およびサーブレットの例を説明しています。それらの 3 つのキュー・マネージャーはすべて、同じ基本 MQSeries Everyplace コンポーネントで構成され、それぞれに固有の特性を付与するものがいくらか加えられています。MQSeries Everyplace には、多くの共通機能をカプセル化するクラス MQeQueueManagerUtils があります。

すべての例は、始動時にパラメーターを必要とします。これらのパラメーターは、標準 ini ファイルに保管されています。 ini ファイルは読み取ることができ、そのデータは MQeFields オブジェクトに変換されます。これについては、31ページの『第3章 MQeFields』を参照してください。 MQeQueueManagerUtils クラス内の **loadConfigFile()** メソッドがこの機能を実行します。

クライアント・キュー・マネージャー

一般に、あるデバイス・プラットフォーム上のアプリケーションがキュー・マネージャーを使用する場合、そのデバイス上ではクライアントが実行します。クライアントは、他のキュー・マネージャーへの多数の接続を開くことができ、対等チャネルを使用するよう構成されていれば、他のキュー・マネージャーからの着信要求を受信することができます。

一般に、サーバーは長時間実行しますが、クライアントは必要に応じて、それらを使用するアプリケーションにより開始および停止されます。複数のアプリケーションがクライアントを共有する場合は、それらのアプリケーションがクライアントの開始および停止を調整する必要があります。

次に示すのは、一般的なクライアントの始動 ini ファイルの例です。

```
*
* ExamplesMQeClient.ini
*   An example ini file for a simple MQe client
*
[Alias]
*
*    Event log class
*
(ascii)EventLog=examples.log.LogToDiskFile
*
*    Network adapter class
*
(ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
*
*    Queue Manager class
*
(ascii)QueueManager=com.ibm.mqe.MQeQueueManager
*
*    Trace handler (if any)
*
(ascii)Trace=examples.trace.MQeTrace
*
*    Message Log file interface
*
(ascii)MsgLog=com.ibm.mqe.adapters.MQeDiskFieldsAdapter
*
*    Class name for File registry
*
(ascii)FileRegistry=com.ibm.mqe.registry.MQeFileSession
*
*    Class name for Private registry
*
(ascii)PrivateRegistry=com.ibm.mqe.registry.MQePrivateSession
*
*    Default Channel class
*
(ascii)DefaultChannel=com.ibm.mqe.MQeChannel
*
```

```
*    Default Transporter class
*
(ascii)DefaultTransporter=com.ibm.mqe.MQeTransporter
*
*    Channel Attribute Rules
*
(ascii)ChannelAttrRules=examples.rules.AttributeRule
*
*    Name of Base Key
*
(ascii)AttributeKey_1=com.ibm.mqe.MQeKey
*
*    Name of Shared Key
*
(ascii)AttributeKey_2=com.ibm.mqe.attributes.MQeSharedKey
*------------------------------------------------------------------------*
*
*  Registry ( configuration data store )
*
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=FileRegistry
*
* Location of the registry
*   (Only use relative directory for development/demo)
*
(ascii)DirName=.¥ExampleQM¥Registry¥
*------------------------------------------------------------------------*
*
*  Queue manager details
*
[QueueManager]
*
*   Name for this Queue Manager
*
(ascii)Name=ExampleQM
```

別名

[Alias] セクションには、別名を設定できる場所があります。 (別名の詳細については、52ページの『別名の使用』を参照してください。)

別名は、等号の左側にあり、完全なクラス名は右側にあります。たとえば、この例では、examples.awt.AwtMQeTrace の代わりに、"Trace" という名前を使用することができます。別名の前の "(ascii)" は、エントリーのタイプ (この場合は、ASCII ストリング) を示します。

別名リストには、ソリューションの所有するクラスを組み込むことができます。

別名リストは、キュー・マネージャー自体によっては処理されません。別名のいくつかはキュー・マネージャーが正常に活動化するために必要とされるため、この別名リスト

はキュー・マネージャーの活動化の前に処理されていなければなりません。たとえば、キュー・ストア・アダプターでは、キューがメッセージを保持するためのストレージ域を持つことができるように定義する必要があります。 MsgLog は、デフォルトのキュー・ストア・アダプターであり、これが存在しない場合には、MsgLog not found 例外が出されます。

キュー・マネージャーの MQeRegistry パラメーター

ini ファイルの [Registry] セクションには、キュー・マネージャーレジストリーのタイプおよびロケーションに関する情報があります。

レジストリーは、キュー・マネージャーに関連した情報のための基本ストアです。キュー・マネージャーごとに 1 つのレジストリーがあります。すべてのキュー・マネージャーは、レジストリーを使用して、次のものを保持します。

- キュー・マネージャー構成データ
- キュー定義
- リモート・キュー定義
- リモート・キュー・マネージャー定義
- ユーザー・データ (構成に依存するセキュリティー情報を含む)

レジストリー・タイプ:

MQeRegistry.LocalRegType (ASCII)
: 開いているレジストリーのタイプ。現在では、*ファイル・レジストリー* および *使用レジストリー* がサポートされています。いくつかのセキュリティー機能では、私用レジストリーが必須です。 221ページの『第8章 セキュリティー』を参照してください。

ファイル・レジストリーの場合、このパラメーターは次のように設定する必要があります。

```
com.ibm.mqe.registry.MQeFileSession
```

私用レジストリーの場合、次のように設定する必要があります。

```
com.ibm.mqe.registry.MQePrivateSession
```

これらの値を表すのに、別名を使用することができます。

ファイル・レジストリー・パラメーター: ファイル・レジストリーには次のパラメーターが必要です。

MQeRegistry.DirName (ASCII)
: レジストリー・ファイルを保持するディレクトリーの名前。

私用レジストリー・パラメーター: 私用レジストリーの場合、次のパラメーターを使用することができます。

MQeRegistry.DirName (ASCII)
レジストリー・ファイルを保持するディレクトリーの名前。

MQeRegistry.PIN (ascii)
私用レジストリー用の PIN。

MQeRegistry.KeyRingPassword (ascii)
レジストリーの秘密鍵を保護するために使用されるパスワードまたは句。

MQeRegistry.CAIPAddrPort (ascii)
ミニ認証サーバーのアドレスおよびポート番号。

MQeRegistry.CertReqPIN (ascii)
レジストリーが信任状を取得できるようにするため、ミニ認証管理者によって事前に割り振られる認証要求番号。

最初の 3 つのパラメーターは常に必要です。最後の 2 つのパラメーターは、ミニ認証サーバーから信任状を取得したい場合に、レジストリーの自動登録のために必要とされるものです。

注: セキュリティー上の理由から、 *PIN* および *KeyRingPassword* (提供されている場合) は、キュー・マネージャーがアクティブにされるとすぐに始動パラメーターから削除されます。

どのタイプのレジストリーの場合でも、デフォルト以外の区切り文字を使用する場合には、これらに加えて *MQeRegistry.Separator (ascii)* が必要です。区切り文字は、項目名のコンポーネントとコンポーネントの間に挿入される文字です。たとえば次のように使用します。

```
<QueueManager><Separator><Queue>
```

このパラメーターはストリングとして指定されますが、単一文字でなければなりません。複数の文字が含まれている場合には、最初の文字だけが使用されます。

レジストリーを開くたびに、毎回同じ区切り文字を使用する必要があります。一度レジストリーが使用されたなら、これを変更することはできません。

このパラメーターが指定されていない場合は、区切り文字にデフォルトの "+" が使用されます。

クライアント・キュー・マネージャーの開始

クライアント・キュー・マネージャーの開始には、次の事柄が関係しています。

1. すでに実行しているクライアントがないことを確認します。(Java 仮想計算機 1 台につき 1 つのクライアントだけが許可されます。)
2. システムに別名を追加します。
3. 必要であれば、トレースを使用可能にします。

4. キュー・マネージャーを開始します。

次のコード断片は、クライアント・キュー・マネージャーを開始します。

```
/*--------------------------------------------------------------------------*/
/* Init - first stage setup                                                 */
/*--------------------------------------------------------------------------*/
public void init( MQeFields parms ) throws Exception
{
  if ( queueManager != null )               /* One queue manager at a time   */
  {
    throw new Exception( "Client already running" );
  }
  sections = parms;                         /* Remember startup parms        */
  MQeQueueManagerUtils.processAlias( sections ); /* set any alias names      */

// Uncomment the following line to start trace before the queue manager is started
//  MQeQueueManagerUtils.traceOn("MQeClient Trace", null); /* Turn trace on   */

  /* Display the startup parameters                                         */
  System.out.println( sections.dumpToString( "#1¥t=¥t#2¥r¥n" ) );

  /* Start the queue manager                                                */
  queueManager = MQeQueueManagerUtils.processQueueManager( sections, null );
}
```

クライアントが一度開始されると、キュー・マネージャー・オブジェクトへの参照は、静的クラス変数 *MQeClient.queueManager* から、または静的メソッド **MQeQueueManager.getReference(queueManagerName)** を使用することによって、取得することができます。

次のコード断片は、システムに別名をロードします。

```
public static void processAlias( MQeFields sections ) throws Exception
{
  if ( sections.contains( Section_Alias ) )   /* section present ?          */
  {                                           /* ... yes                    */
    MQeFields section = sections.getFields( Section_Alias );
    Enumeration keys  = section.fields( );    /* get all the keywords       */
    while ( keys.hasMoreElements() )          /* as long as there are keywords*/
    {
      String    key   = (String) keys.nextElement();     /* get the Keyword   */
      MQe.alias( key, section.getAscii( key ).trim( ) ); /* add               */
    }
  }
}
```

各別名をシステムに追加するには、**processAlias** メソッドを使用します。 MQSeries Everyplace とアプリケーションのどちらもロードされると、別名を使用することができます。 MQSeries Everyplace が正しく機能するためには、 58ページの ini ファイルで示されたたくさんの別名が必要になるので、それらを除去しないでください。

キュー・マネージャーの開始には、次の事柄が関係しています。

1. キュー・マネージャーのインスタンス化。ロードするキュー・マネージャー・クラスの名前は、別名 QueueManager で指定されます。クラスをロードし、空コンストラクターを呼び出すには、 MQSeries Everyplace クラス・ローダーを使用してください。
2. **activate** オブジェクトを使用し、 ini ファイルの MQeFields オブジェクト表現を渡すことによって、キュー・マネージャーをアクティブにする。キュー・マネージャーは、始動パラメーターから [QueueManager] および [Registry] セクションだけを使用します。

次のコード断片は、キュー・マネージャーを開始します。

```
public static MQeQueueManager processQueueManager( MQeFields sections,
 Hashtable ght ) throws Exception
{                                                 /*                          */
  MQeQueueManager queueManager = null;            /* work variable             */
  if ( sections.contains( Section_QueueManager) )     /* section present ?     */
  {                                               /* ... yes                  */
    queueManager = (MQeQueueManager) MQe.loader.loadObject(Section_QueueManager);
    if ( queueManager != null )                   /* is there a Q manager ?    */
    {
      queueManager.setGlobalHashTable( ght );
      queueManager.activate( sections );          /* ... yes, activate         */
    }
  }
  return( queueManager );                         /* return the alloated mgr   */
}
```

MQePrivateClient の例

MQePrivateClient は、MQeClient の拡張で、キュー・マネージャーとレジストリーをセキュア・キューで使用できるように構成する機能が追加されています。セキュア・クライアントの場合、始動パラメーターの [Registry] セクションは、次のように拡張されています。

```
* Extract from MQePrivateClient.ini
*
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=PrivateRegistry
*
*    Location of the registry
*
(ascii)DirName=.¥ExampleQM¥PrivateRegistry
*
* PIN
*
(ascii)PIN=not set
*
* Certificate request pin
*
(ascii)CertReqPIN=not set
*
```

```
* Key ring password
*
(ascii)KeyRingPassword=not set
*
* Network address of certificate authority
*
(ascii)CAIPAddrPort=9.20.7.219:8082er
```

これらのフィールドについては、60ページの『キュー・マネージャーの MQeRegistry パラメーター』を参照してください。セキュア・キューおよび MQePrivateClient についての詳細は 221ページの『第8章 セキュリティー』を参照してください。

MQePrivateClient (および MQePrivateServer) を作動させるには、始動パラメーターに *CertReqPIN*、*KeyRingPassword* および *CAIPAddrPort* が含まれていてはなりません。MQSeries Everyplace 標準版を使用した MQePrivateClient のレジストリー・セクションは、次のようになります。

```
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=PrivateRegistry
*
*   Location of the registry
*
(ascii)DirName=.¥ExampleQM¥PrivateRegistry
*
* PIN
*
(ascii)PIN=not set
```

サーバー・キュー・マネージャー

通常、サーバーはサーバー・プラットフォーム上で実行します。サーバーはサーバー側のアプリケーションを実行できますが、クライアント側のアプリケーションも実行できます。クライアントの場合と同じように、サーバーは、サーバーとクライアントの両方で多くの他のキュー・マネージャーへの接続を開くことができます。クライアントと異なるサーバーの主な特性の 1 つは、たくさんの同時着信要求を処理できるという点です。しばしば、サーバーは、多くのクライアントが MQSeries Everyplace ネットワークへ入るためのエントリー・ポイントとして働きます。MQSeries Everyplace は、次のようなサーバーの例を提供しています。

MQeServer
コンソール・ベースのサーバー

MQePrivateServer
拡張セキュリティーを持つ、コンソール・ベースのサーバー

AwtMQeServer
MQeServer へのグラフィカル・フロントエンド

MQBridgeServer

このサーバーは、通常のMQSeries Everyplace サーバー機能に加えて、MQSeries ファミリーの他のメンバーとメッセージの送受信を行うことができます。このサーバーは、パッケージ examples.mqbridge.queuemanager の中にあります。これについては、173ページの『第7章 MQSeries-ブリッジ』を参照してください。

MQeServer の例

MQeServer のサーバー・インプリメンテーションは最も単純です。

このサーバーは、次のコマンドによって開始することができます。

```
<javaCommand> examples.queuemanger.awt.MQeServer  <startupIniFile>
```

この場合

javaCommand

Java アプリケーションを開始するために使用されるコマンド (たとえば、**java**)

startupIniFile

キュー・マネージャーおよびサーバーの始動パラメーターを含む ini ファイル (たとえば、.¥ExamplesMQeServer.ini)

バッチ・ファイル ExamplesMQeServer.bat は、 ini ファイル .¥ExamplesMQeServer.ini を使ってサーバーを開始するためのショートカットを提供します。クライアント・キュー・マネージャーの場合と同様、 ini ファイルを使用して、サーバーの始動パラメーターを保持します。サーバー・キュー・マネージャーの場合は、標準クライアント・キュー・マネージャーの ini ファイルを拡張し、 [ChannelManager] および [Listener section] を組み込む必要があります。サーバー始動パラメーターを拡張した典型的な例は、次のとおりです。

```
* Extract from ExamplesMQeServer.ini
*
[ChannelManager]
*
*   Maximum number of channels allowed
*
(int)MaxChannels=0
*-------------------------------------------------------------------------*
[Listener]
*
*   FileDescriptor for listening adapter
*
(ascii)Listen=Network::8082
*
*   FileDescriptor for Network read/write
*
(ascii)Network=Network:
```

```
*
*   Channel time-out interval in seconds
*
(int)TimeInterval=300
```

2 つのキュー・マネージャーが相互に通信するとき、MQSeries Everyplace はその 2 つのキュー・マネージャーの間のチャネルを開きます。チャネルは、キュー・マネージャー・パイプに対するキュー・マネージャーとして使用される論理エンティティーです。いつでも複数のチャネルを開くことができます。

ini ファイルの新規セクションは、チャネルの使用法を制御します。 ChannelManager セクションの *MaxChannels* パラメーターは、いつでも開くことのできるチャネルの最大数を制御します。 特殊値が 0 であるということは、キュー・マネージャーが無制限の数のチャネルを処理できるということを意味しています。Listener セクションには、着信ネットワーク要求が処理される方法に関するパラメーターが含まれています。

Listen
: 着信ネットワーク要求を処理するネットワーク・アダプター。たとえば、http アダプターまたは純正の tcp/ip アダプター。アダプター名だけでなく、アダプターが listen する方法を示すパラメーターも渡すことができます。たとえば、Listen=Network::8082 は、Network アダプターを使用することを意味します。ここで、Network は、ポート 8082 上で listen する別名です。(これは、Network 別名が HTTP または tcp/ip アダプターのいずれかに設定されていることを前提としています。)

Network
: このパラメーターを使用して、初期ネットワークが受け入れられたら、ネットワーク読み取り / 書き込み要求に使用されるアダプターを指定します。通常、これは *Listen* パラメーターで使用されるアダプターと同じです。

TimeInterval
: 使用されていないチャネルがタイムアウトになるまでの時間 (秒)。チャネルは、単一のキュー・マネージャー要求よりも長く持続する永続的な論理エンティティーであり、ネットワークの破損後も存続することが可能なので、一定の時間、活動状態にないチャネルをタイムアウトにしなければならない場合があります。

MQeServer を作成するのは、 MQeClient を作成した後です。

1. サーバー始動パラメーターを **init** メソッドに渡します。
2. JVM ごとに 1 台のサーバーだけが実行されるように検査します。
3. 別名をロードし、必要であればトレースを有効にします。

次のコードは、サーバーを開始するときに使用される **init** メソッドを示しています。

```
public void init( MQeFields parms ) throws Exception
{
  if ( initialized )                          /* Only one server at a time     */
    throw new Exception( "Server already running" );
```

```
  sections = parms;                          /* Remeber startup parms         */
  MQeQueueManagerUtils.processAlias( sections ); /* set any alias names       */

// Uncomment the following line to start trace before the queue manager is started
//  MQeQueueManagerUtils.traceOn("MQeServer Trace", null); /* Turn trace on   */

  /* Display the startup parameters                                           */
  System.out.println( sections.dumpToString( "#1¥t=¥t#2¥r¥n" ) );
}
```

サーバーが初期化されたら、パラメーターを *true* に設定し、**activate** メソッドを使って、サーバーをアクティブにしてください。アクティブになったら、パラメーターを *false* に設定し、**activate** メソッドを呼び出して、そのサーバーを非活動化することができます。

サーバーを活動化すると、次の事柄が生じます。

1. チャネル・マネージャーが開始されます。
2. ユーザー指定のクラスが追加してロードされ、空コンストラクターが呼び出されます。
3. キュー・マネージャーが開始されます。
4. チャネル・リスナーが開始されます。

これは、次のコードで示されます。

```
public void activate( boolean Start ) throws Exception
{
  if ( Start )                                 /* activate ?                  */
  {                                            /* ... yes                   */
    if ( ! initialized )                       /* been here before            */
    {                                          /* ... no                      */
      /* allocate Chan Mgr                                                    */
    channelManager  = MQeQueueManagerUtils.processChannelManager( sections );

      /* assign any class aliases                                           */
    MQeQueueManagerUtils.processAlias( sections );

      /* check for any pre-loaded classes                                   */
    loadTable = MQeQueueManagerUtils.processPreLoad( sections );
      initialized = true;                      /* only once                   */
    }                                          /*                             */
    /* setup and activate the queue manager                                 */
    queueManager = MQeQueueManagerUtils.processQueueManager( sections,
   channelManager.getGlobalHashtable( ) );

    /* setup and activate the listener for incomming connections            */
    channelListener = MQeQueueManagerUtils.processListener(
   sections, channelManager );
  }
  else                                         /* ... no                      */
  {                                            /*                             */
    if ( channelListener != null )  channelListener.stop( );
    if ( queueManager    != null )  queueManager.close( );
```

```
    channelListener = null;                  /* release object              */
    queueManager    = null;                  /* release object              */
  }
}
```

リスナーが開始されると、サーバーがネットワーク要求を受け入れる準備が整います。

サーバーが非活動化されると、次のことが生じます。

1. チャネル・リスナーが停止し、新規着信要求が妨げられます。
2. キュー・マネージャーがクローズされます。

MQeQueueManagerUtils クラスのコードの次のセクションは、それぞれのコンポーネントを処理します。

チャネル・マネージャーを開始するセクションは次のとおりです。

```
public static MQeChannelManager processChannelManager( MQeFields sections )
throws Exception
{
  MQeChannelManager channelManager = null;    /* work variable               */
  if ( sections.contains( Section_ChannelManager ) )  /* section present ?    */
  {                                           /* ... yes                    */
    MQeFields section = sections.getFields( Section_ChannelManager );
    channelManager    = new MQeChannelManager( );     /* load the manager     */
    channelManager.numberOfChannels( section.getInt( "MaxChannels" ) );
  }                                           /*                             */
  return( channelManager );                   /* return the allocated mgr    */
}
```

このメソッドは、チャネル・マネージャーをインスタンス化した後、始動パラメーターの [ChannelManager] セクションの *MaxChannels* パラメーターを使って、許可されるチャネルの最大数を設定します。

また、キュー・マネージャーがロードされるときにロードするクラスのセットを指定することもできます。これらは ini ファイルの [PreLoad] セクションに追加されます。エントリーは次の例に示すとおり、(ascii)uniqueName=class という形式でなければなりません。

```
[PreLoad]
*
*   Classes to load at server startup
*
(ascii)StartClass1=test.ServerTest1
(ascii)StartClass2=test.ServerTest2
```

The following section of code loads the preload classes:

```
public static Hashtable processPreLoad( MQeFields sections ) throws Exception
{
  Hashtable loadTable = new Hashtable();      /* allocate load table         */
  if ( sections.contains( Section_PreLoad ) ) /* section present ?            */
```

```
  {                                          /* ... yes                    */
    MQeFields section = sections.getFields( Section_PreLoad );
    Enumeration keys  = section.fields();    /* get all the keywords       */
    while ( keys.hasMoreElements() )         /* as long as there are keywords*/
    try                                      /* incase of error            */
    {                                        /*                            */
      String key = section.getAscii( (String) keys.nextElement() ).trim( );
      loadTable.put( key, MQe.loader.loadObject( key ) );
    }
    catch ( Exception e )                    /* error occured              */
    {
      e.printStackTrace();                   /* show the error             */
    }
  }
  return( loadTable );                       /* return the table           */
}
```

ini ファイルの [PreLoad] セクションで指定されるそれぞれのクラスごとに、次のことが生じます。

1. MQeLoader を使用して、クラスがロードされます。これはクラスの空コンストラクターを呼び出すので、初期化コードまたは始動コードは、このコンストラクターに入れる必要があります。
2. ロードされると、クラスの参照がハッシュ・テーブルに置かれます。このテーブルは、サーバー内の他のメソッドでも使用できます。たとえば、サーバーがクローズするときに、サーバーの **close** メソッドを拡張して、プリロードされたすべてのクラスの **close** メソッドを実行することができます。

MQePrivateServer の例

MQePrivateServer は、 MQeServer の拡張で、キュー・マネージャーとレジストリーをセキュア・キューで使用できるように構成する機能が追加されています。 221ページの『第8章 セキュリティー』を参照してください。

AwtMQeServer の例

パッケージ examples.awt にある AwtMQeServer は、コンソール・ベースのサーバーに対するグラフィカル・フロントエンドを提供します。

このサーバーは、次のコマンドで開始します。

```
<javaCommand> examples.awt.AwtMQeServer  <startupIniFile>
```

Where:

javaCommand
: Java アプリケーションを開始するために使用されるコマンド (たとえば、**java**)

startupIniFile
: キュー・マネージャーおよびサーバーの始動パラメーターを含む ini ファイル (たとえば、.¥ExamplesAwtMQeServer.ini)

バッチ・ファイル ExamplesAwtMQeServer.bat は、ファイル .¥ExamplesAwtMQeServer.ini を使ってサーバーを開始するためのショートカットを提供します。

AwtMQeServer は、次の別名を追加して使用します。

サーバー
: このクラスがグラフィカル・フロントエンドを提供するサーバー・クラス。

「管理 (Admin)」
: 管理コンソールを提供するクラスの名前。

ファイル例 .¥ExamplesAwtMQeServer.ini は、次のように別名を設定します。

```
*
*    Admin console (if any)
*
(ascii)Admin=examples.administration.console.Admin
*
*    Base Server class
*
(ascii)Server=examples.queuemanager.MQeServer
```

専用サーバーが開始すると、次のウィンドウが表示されます。

図 8. AWT MQSeries Everyplace サーバー・ウィンドウ

ボタンの機能は次のとおりです。

「終了 (Exit)」
: サーバーをクローズし、**System.exit()** を実行します。

「停止| 実行 (Stop| Run)」
: サーバーが実行中の場合、**「停止 (Stop)」**はそれを停止します。サーバーが停止すると、ボタンは**「実行 (Run)」**を表示します。

  開始は、次のコードによって実行されます。

```
if ( running )                    /* running ?                    */
{
  setText( North, index, "Run"  );        /* ... yes,             */
  server.activate( false );               /* stop server          */
}
else
```

```
{
  setText( North, index, "Stop" );        /* ... no,  i.e start   */
  if ( server == null )                   /* initialized before ? */
  {                                       /* yes,                 */
    /* Load the startup parms and setup class aliases             */
    MQeFields sections
      = MQeQueueManagerUtils.loadConfigFile( iniName );
    MQeQueueManagerUtils.processAlias( sections );
    /* Load the server and initialize if first pass               */
    server = (MQeServer)MQe.loader.loadObject( "Server" );
    server.init( sections );
  }
  server.activate( true );        /* Activate the server          */
}
running = ! running;              /* change state                 */
```

「トレース (Trace)」

トレースを活動化または非活動化します。これは、次のコードによって実行されます。

```
/* Get current trace handler if any ..                           */
MQeTraceInterface    trace = MQe.getTraceHandler( );
if ( trace == null )              /* If trace is not on,start it  */
  MQeQueueManagerUtils.traceOn( this.getTitle() + " - Trace", null );
else                              /* otherwise stop it            */
  MQeQueueManagerUtils.traceOff();
```

「管理 (Admin)」

管理コンソールを開始または停止します。次のコードがこの機能をインプリメントします。

```
if ( adminGUI != null && adminGUI.active )
{                                 /* GUI active so                */
  adminGUI.close();               /* close it                     */
  adminGUI = null;
}
else if ( adminGUI == null ||
        ( adminGUI != null && !adminGUI.active ) )
{                                 /* GUI not running or not active*/
  adminGUI = (Admin)MQe.loader.loadObject( "Admin" );
  adminGUI.activate();            /* so load and actiate it       */
}
```

「ヘルプ (Help)」

ダイアログについて表示します。

さらに、イベント・ロギングをオン / オフに切り替えて、使用するロガーをドロップダウン・リスト・ボックスから選択することができます。次の選択が可能です。

- ログ記録しない
- examples.eventlog.LogToDiskFile
- examples.eventlog.LogToNTEventLog

サーブレット

キュー・マネージャーは、スタンドアロン・サーバーとして実行するだけでなく、サーブレット内にカプセル化することによって、Web サーバー内で実行することができます。サーブレット・キュー・マネージャーは、サーバー・キュー・マネージャーとほぼ同じ機能を持ちます。MQeServlet はサーブレットのインプリメンテーション例を提供します。サーバーの場合と同様、サーブレットは ini ファイルを使用して始動パラメーターを保持します。サーブレットはサーバーと同じ MQSeries Everyplace コンポーネントを多数使用し、サーバー ini ファイルを使用することができます。

サーブレットで必要でない主なコンポーネントはチャネル・リスナーで、この機能は Web サーバー自体が実行します。 Web サーバーは HTTP データ・ストリームしか処理しないので、MQSeries Everyplace クライアント サーブレットと通信する MQSeries Everyplace クライアントは、HTTP アダプターを使用する必要があります (com.ibm.mqe.adapters.MQeTcpipHttpAdaper)。サーブレット内で実行しているキュー・マネージャーへの接続を構成するとき、サーブレットの名前は、接続のパラメーター・フィールドで指定する必要があります。次の定義では、サーブレット /servlet/MQSeries Everyplace で、キュー・マネージャー PayrollQM への接続を構成します。

接続名 PayrollQM

チャネル
com.ibm.mqe.MQe

チャネル・アダプター
com.ibm.mqe.adapters.MQe

TcpipHttpAdaper
192.168.0.10:80

パラメーター
/servelet/MQe

オプション

あるいは、関係のある別名が設定されている場合は、次のように接続を構成することができます。

接続名 PayrollQM

チャネル
DefaultChannel

アダプター
Network:192.168.0.10:80

パラメーター
/servelet/MQe

オプション

Web サーバーは複数のサーブレットを実行することができます。 Web サーバー内では複数の異なる MQSeries Everyplace サーブレットを実行することができます。ただし、次のような制限があります。

- それぞれのサーブレットには固有の名前が必要です。
- 1 つのサーブレットあたり 1 つのキュー・マネージャーだけが許可されます。
- それぞれの MQSeries Everyplace サーブレットは、別々の Java 仮想計算機 (JVM) で実行しなければなりません。

MQSeries Everyplace サーブレットは、javax.servlet..http.HttpServlet を拡張したものであり、新規要求を開始、停止、および処理するためのメソッドをオーバーライドします。次のコード断片は、サーブレットを開始します。

```
/**
 * Servlet Initialisation......
 */
public void init(ServletConfig sc) throws ServletException
{
  // Ensure supers constructor is called.
  super.init(sc);

  try
  {
    // Get the the server startup ini file
    String startupIni;
    if ( ( startupIni = getInitParameter("Startup")) == null )
      startupIni = defaultStartupInifile;

    // Load it
    MQeFields sections = MQeQueueManagerUtils.loadConfigFile(startupIni);

    // assign any class aliases
    MQeQueueManagerUtils.processAlias( sections );

    // Uncomment the following line to start trace before the queue
    // manager is started
    //      MQeQueueManagerUtils.traceOn("MQeServlet Trace", null);

    // Start channel manager
    channelManager  = MQeQueueManagerUtils.processChannelManager( sections );

    // check for any pre-loaded classes
    loadTable = MQeQueueManagerUtils.processPreLoad( sections );

    // setup and activate the queue manager
    queueManager = MQeQueueManagerUtils.processQueueManager( sections,
   channelManager.getGlobalHashtable( ) );

    // Start ChannelTimer  (convert time-out from secs to millsecs)
    int tI =
      sections.getFields(MQeQueueManagerUtils.Section_Listener).getInt( "TimeInterval" );
    long timeInterval = 1000 * tI;
    channelTimer = new MQeChannelTimer( channelManager, timeInterval );

    // Servlet initialisation complete
    mqe.trace( 1300, null );
  }
  catch (Exception e)
  {
    mqe.trace( 1301, e.toString() );
    throw new ServletException( e.toString() );
  }
}
```

サーバー始動と比較した場合の主な違いは、次のとおりです。

- サーブレットはスーパークラスの **init** メソッドをオーバーライドします。このメソッドは、Web サーバーがサーブレットを開始するために呼び出されます。一般にこれが生じるのは、サーブレットの最初の要求が到着したときです。
- 始動パラメーター ini ファイルの名前は、コマンド行から渡すことができません。例では、サーブレット・メソッド **getInitParameter()** を使用して、その名前を取得することになっています。このメソッドは、パラメーターの名前を取得して、値を戻すものです。 MQSeries Everyplace サーブレットは、ini ファイル名が入っていると予期される、 *Startup* パラメーターを使用します。 Web サーバー内でパラメーターを構成するメカニズムは、Web サーバーに依存しています。
- Web サーバーがサーブレットのためにすべてのネットワーク要求を処理するので、チャネル・リスナーは開始されません。
- チャネル・リスナーが存在しないため、長時間活動状態にないチャネルをタイムアウトにするためのメカニズムが必要です。この機能を実行するために、単純なタイマー・クラス MQeChannelTimer がインスタンス化されます。 *TimeInterval* 値が、ini ファイルの [Listener] セクションから使用される唯一のパラメーターです。

サーブレットは着信要求の受信および処理を Web サーバーに依存しています。 Web サーバーが、要求が MQSeries Everyplace サーブレットに対するものであると判断すると、**doPost()** メソッドを使って、要求を MQSeries Everyplace に渡します。次のコードがこの要求を処理します。

```
/**
 * Handle POST......
 */
public void doPost(HttpServletRequest request,
                   HttpServletResponse response) throws IOException
{
  // any request to process ?
  if (request == null)
    throw new IOException("Invalid request");
  try
  {
    int max_length_of_data = request.getContentLength();     // data length
    byte[] httpInData = new byte[max_length_of_data];        // allocate data area
    ServletOutputStream httpOut = response.getOutputStream();// output stream
    ServletInputStream  httpIn  = request.getInputStream();  // input stream

    // get the request
    read( httpIn, httpInData, max_length_of_data);

    // process the request
    byte[] httpOutData = channelManager.process(null, httpInData);

    // appears to be an error in that content-length is not being set
    // so we will set it here
    response.setContentLength(httpOutData.length);
    response.setIntHeader("content-length", httpOutData.length);

    // Pass back the response
    httpOut.write(httpOutData);
  }
  catch (Exception e)
  {
```

```
    // pass it on ...
    throw new IOException( "Request failed" + e );
  }
}
```

このメソッドには、以下の処理が含まれます。

1. HTTP 入力データ・ストリームを*バイト配列*に読み取ります。入力データ・ストリームをバッファーに入れ、**read()** メソッドによって、処理を継続する前にデータ・ストリーム全体が確実に読み取られるようにすることもできます。

   注: MQSeries Everyplace は **doPost()** メソッドによってのみ要求を処理します。**doGet()** メソッドを使用して要求を受け入れることはありません。
2. 要求はチャネル・マネージャーを介して MQSeries Everyplace に渡されます。この時点から、要求のすべての処理は、キュー・マネージャーなどのコア MQSeries Everyplace クラスによって処理されます。
3. MQSeries Everyplace が要求の処理を完了すると、http ヘッダーにバイト配列として含められた結果を戻します。このバイト配列は、Web サーバーに渡され、要求を発信したクライアントに転送されます。

基本クラスを使用したキュー・マネージャーの構成

キュー・マネージャーを作成したり削除したりするには、MQeQueueManagerConfigure を使用することをお勧めしますが、このセクションでは、同じ機能を基本クラスから作成する方法を説明します。

キュー・マネージャーの活動化

キュー・マネージャーをアクティブにするには、次のものが必要です。

- 事前に構成されたレジストリー
- レジストリーをアクティブにする方法をキューマネージャーに知らせる、活動化パラメーターのセット

キュー・マネージャーが活動化されると、活動化パラメーターがそれに渡されます。これらのパラメーターは、別の MQeFields オブジェクトの内部に組み込まれた MQeFields オブジェクトから成っています。

使用される組み込み MQeFields オブジェクトの名前は、 MQeQueueManager クラスで定義されます。

MQeQueueManager.QueueManager

活動化されるキュー・マネージャーの名前。

MQeQueueManager.Registry

キュー・マネージャーの事前定義されたレジストリーの場所。

MQeQueueManagerUtils.Section_Aliases
MQSeries Everyplace 別名

レジストリーには、キュー・マネージャーが所有するキューの定義、その他の既知のキュー・マネージャーの定義、および構成可能なキュー・マネージャーのセットアップ・データの一部が含まれます。このセットアップ・データは次のものです。

Queue manager description
キュー・マネージャー記述を含むストリング

Queue manager rules
キュー・マネージャーのルールとして使用する、クラスの名前を含むストリング (107ページの『キュー・マネージャー・ルール』を参照)。

Default queue store
デフォルトのキュー・ストア (キューはここにメッセージを保管する) の場所であるパス名。これは、まだキュー・ストア・フィールドを含んでいないキューが、キュー・マネージャーに追加される場合にのみ使用されます。キューの名前は、デフォルトのストリングに追加されて、固有のキュー・ストア・パス名をキューに与えます。

Channel attribute rules
チャネル属性ルールとして使用されるクラスの名前を含むストリング。これらのルールは、非空属性を持つリモート・キューを処理する際の動作方法を定義します。

Channel Timeout
チャネル・タイムアウトとなる長精度の値 (ミリ秒単位)。 2 つのキュー・マネージャー間のチャネルが、この期間よりも長く使用されていない場合には、チャネルがクローズされます。

これらの値はすべて MQSeries Everyplace 管理 (119ページの『第6章 メッセージング・リソースの管理』を参照) を使って更新したり、またキュー・マネージャーの作成時に構成したりすることができます。

このMQSeries Everyplace 別名については、52ページの『別名の使用』で詳しく解説されています。

MQSeries Everyplace は、事前定義された構成でキュー・マネージャーを開始する 2 つのクラスを提供します。(これらのクラスは `examples` ディレクトリーにあります。)

MQeClient
キュー・マネージャーを、クライアントとして開始します。

MQeServer
キュー・マネージャーを、MQSeries Everyplace サーバーの一部として開始します。

必要な処理はすべて、キュー・マネージャーが開始する前に、これらのクラスによって処理されます。

これらのクラスのどちらも使用せずに、別名リストを処理してキュー・マネージャーを活動化することができます。別名リストは、**MQe.alias** メソッドを使って処理されます。下の例では、別名 "Trace" が examples.awt.AwtMQeTrace に設定されます。

```
alias( "Trace", "examples.awt.AwtMQeTrace" );
```

MQeClient と MQeServer の両方が、キュー・マネージャー・パラメーターを含む .ini ファイルを受け入れます。 .ini ファイルのエントリーは、必要な組み込み MQeFields オブジェクトに変換されます。この処理は、examples.queuemanager.MQeQueueManagerUtils クラスによって行われ、そのクラスは **MQe.alias** メソッドを使って別名リストを処理します。

次のコード断片は、上記の手順を示しています。

```
public static void processAlias( MQeFields sections ) throws Exception
{
  if ( sections.contains( Section_Alias ) )   /* section present ?          */
  {                                           /* ... yes                    */
    MQeFields section = sections.getFields( Section_Alias );
    Enumeration keys  = section.fields( );    /* get all the keywords       */
    while ( keys.hasMoreElements() )          /* as long as there are       */
                                              /* keywords                   */
    {                                         /*                            */
      /* get the Keyword    */
      String key = (String) keys.nextElement();
      /* add alias          */
      MQe.alias( key, section.getAscii( key ).trim( ) );
    }                                         /*                            */
  }                                           /*                            */
}
```

このメソッドへの入力データである MQeFields オブジェクト sections は、MQeFields フォームの ini ファイルです。 ini ファイルは、MQeQueueManagerUtils の **loadConfigFile()** メソッドで、MQeFields オブジェクト・フォームに変換されます (これは **MQeFields.restoreFromString()** メソッドを使用します)。

sections に別名リストが含まれているかを調べるテストが行われます。別名リスト ini ファイル のセクション名は、定数 Section_Alias で定義されます。別名リストが使用可能な場合は、 **getFields()** が sections に対して実行され、別名リスト (MQeFields オブジェクト) を戻します。その後、別名リストの内容が列挙され、コードが列挙全体をループして、それぞれの別名ごとに別名コマンドを呼び出します。

キュー・マネージャーの使用

MQSeries Everyplace アプリケーションおよび Java 仮想計算機

MQSeries Everyplace キュー・マネージャーの Java 版は、Java 仮想計算機 (JVM) のインスタンスの内部で実行します。現在、MQSeries Everyplace は、1 つの JVM につき 1 つのキュー・マネージャーだけしか起動させません。しかし、同じデバイス上で JVM の複数インスタンス (Java コマンドが呼び出されるたびに、新規の Java 仮想計算機が作成される) を起動することができます。つまり、複数の MQSeries Everyplace キュー・マネージャーを同じデバイス上に作成することができます。これらのキュー・マネージャーにはそれぞれ固有の名前が必要であり、固有の名前がなければ、予期しない動作が生じることがあります。

Java MQSeries Everyplace アプリケーションは、それらが使用しているキュー・マネージャーと同じ JVM の内部で実行しなければなりません。これを行う洗練された方法は、アプリケーション・ランチャーを使用することです。これは、別個のスレッド上にあるキュー・マネージャーと多数の MQSeries Everyplace アプリケーションを開始するクラスです。そのようなクラスの例が、次のコード断片に示されています。

```
/* extends from   MQe base class */
public class appLauncher extends   MQe implements Runnable
{
  Thread[] threads    = null; /* thread references */
  String[] appList    = null; /* list of   MQSeries Everyplace apps */
  int      appCount   = 0;
  String lock = new String();
  MQeQueueManager qmgr = null; /* reference to QMgr */

  public static void main( String args[] )
  {

  try
    {
      (new appLauncher()).startApplications();
    }
  catch ( Exception e )
    {
      System.err.println( "Exception on starting applications" );
      e.printStackTrace( System.err );
    }
  }

  public void startApplications( String args[] ) throws Exception
  {
    boolean active = false; /* any active threads? */
    /* create an array of the thread references of the applications  */
    /* being launched */
    threads = new Thread[ args.length ];
    appList = args; /* keep the list of the applications to be launched */
    /* loop through the list of apps being launched & start a new  */
    /* thread for each one */
```

```
  for ( int i = 0; i < appList.length; i++ )
  {
    Thread th = new Thread( this ); /* create a new thread */
    threads[i] = th; /* keep reference */
    th.start(); /* start new thread */
    /* loop until queue manager is active then start rest of apps */
    if ( i == 0 )
      while( qmgr == null );
  }
  /* keep appLauncher thread alive until all other apps have finished */
  while( active )
  {
    active = false;
    /* loop through thread references, starting at element 1 */
    /* remember first element in appList is QMgr ini file path name */
    for( int j=1; j < appList.length; j++ )
      if ( threads[j] != null )
        active = true; /* thread still active */
  }
  if ( qmgr != null )
    qmgr.close(); /* close queue manager */
}

/* this method called for each application being launched, plus the  */
/*  queue manager */
public void run()
{
  int currentApp; /* which element in threads table */
  synchronized( lock )
  {
    currentApp = appCount;
    appCount++; /* update count */
  }
  try
  {
    /* first element is QMgr ini file path name */
    if ( currentApp == 0 ) /* start  queue manager */
    {
      MQeClient client = new MQeClient( appList[0] );
      qmgr = client.queueManager; /* QMgr now active */
    }
    else  /* load application */
                                      /* (this invokes default constructor) */
      loader.loadObject( appList[currentApp] );
  }
  catch ( Exception e )
  {
    e.printStackTrace( System.err );
  }
  finally
  { /* get thread reference for this app */
    Thread th = threads[currentApp];
    threads[currentApp] = null; /* nullify reference */
```

```
      th.stop(); /* stop thread */
    }
  }
}
```

このクラスに提供される引き数は、キュー・マネージャーの ini ファイルのパス名で、その後に、立ち上げられる MQSeries Everyplace アプリケーションのリストが続きます。すべてのアプリケーションは、デフォルトのコンストラクターを使用して呼び出されます。

アプリケーション・ランチャーは、次のコマンドで起動されます。

```
java appLauncher
<ini ファイル・パス名><アプリケーション・クラス名><アプリケーション・クラス名>...
```

たとえば、次のようにします。

```
java appLauncher
           e:¥¥MQe¥¥TestQMgr¥¥TestQMgr.ini examples.queuemanager.TestMQeApp
```

すべてのアプリケーションは、**MQeQueueManager.getReference()** を使用して、すでに JVM 内部で実行しているキュー・マネージャーへのオブジェクト参照を取得する必要があります。

RunList を使ってアプリケーションを立ち上げる

MQSeries Everyplace アプリケーションを立ち上げる代替方法は、RunList メカニズムを使用することです。キュー・マネージャーの活動化パラメーターの一部として、MQSeries Everyplace アプリケーションの 2 つのリスト (*実行リスト* とも言う) を提供することができます。最初のリストには、キュー・マネージャーが活動化された後で立ち上げられるアプリケーションが含まれています。 2 番目のリストには、キュー・マネージャーがクローズ要求を受け取ると立ち上げられるアプリケーションが含まれています。

実行リストに含まれるアプリケーションは、 MQeRunListInterface をインプリメントする必要があります。キュー・マネージャーは インターフェースで定義された **activate()** メソッドを呼び出してアプリケーションを活動化し、これに使用可能なセットアップ情報を渡します。

アプリケーションが MQeRunListInterface をインプリメントしない場合、そのアプリケーションはただ起動されるだけで、セットアップ情報は渡されません。

ini ファイル内の [AppRunList] セクションには、キュー・マネージャーの活動化時に立ち上げられるアプリケーションの名前が含まれています。アプリケーションの記号名は等号の左側にあり、アプリケーションの完全なクラス名はその右側にあります。 キュー・マネージャーの ini ファイル例 に示されているとおりです。

アプリケーションのセットアップ・データは、 [アプリケーションの記号名] という見出しのセクションの中に指定できます。

キュー・マネージャーの ini ファイルの例

```
* Sample  queue manager ini file

*  queue manager setup info
[QueueManager]
* Name for this  queue manager
(ascii)Name=ServerQMgr8082

* Registry setup info
[Registry]
* QueueManager Registry type (ascii)LocalRegType=com.ibm.mqe.registry.MQePrivateSession
* Location of the registry
(ascii)DirName=d:¥development¥Rename¥Classes¥ServerQMgr8082¥Registry
* Registry access PIN
(ascii)PIN=12345678

* List of applications to launched at  queue manager activation-time
[AppRunList]
(ascii)App1=examples.queuemanager.TestMQeApp
(ascii)App2=examples.administration.AdminApp

* Setup info for App1 - the data in this section is passed to the application
[App1]
(ascii)DefaultMsgPriority = 7
(long)Timeout = 30000

* Setup info for App2 - the data in this section is passed to the application
[App1]
(ascii)DefaultQueueName=AdminReplyQueue
```

キュー・マネージャーの活動化時に起動されるアプリケーションは、キュー・マネージャーが活動化を継続できるように、できるだけ早くキュー・マネージャーに制御を戻す必要があります。アプリケーションが長時間実行されるタスクの場合は、呼び出されたものとは別のスレッドを使って初期化する必要があります。アプリケーションは、自分が作成するスレッドを管理する責任があります。

キュー・マネージャーの **close** 時に呼び出されるアプリケーションが戻らない場合、キュー・マネージャーはシャットダウンしません。

キュー・マネージャーの活動化時に立ち上げられるアプリケーションの例

```
public class ExampleApp extends   MQe implements MQeRunListInterface,
                                                  Runnable,
                                                  MQeMessageListenerInterface
{
  Thread th = null;
  MQeQueueManager qmgr = null;
  ...
  /*Called by the  queue manager to activate the application          */
  public Object activate( Object owner, Hashtable loadTable,
                          MQeFields setupData )
  {
    qmgr = (MQeQueueManager)owner;   /*QMgr is owner of the application*/
    processSetupData( setupData );      /*Process the setup information*/
    th = new Thread( this );            /*Create a new thread to listen*/
```

```
    th.start();                              /*for incoming messages*/
    return (null);                        /*return control to the QMgr*/
  }
  public void run()
  {
  try
    {
       /*Create a message listener for incoming messages*/
       qmgr.addMessageListener( this, "MyQueue", null );
       /* Loop indefintely keeping application alive */
       while( true );
    }
    catch ( Exception e )
    {
      e.printStackTrace( System.err );
    }
  }
  ...
}
```

この例では、**activate()** メソッドを使用してアプリケーションが起動されます。このメソッドは、そのセットアップ・データを処理し、別個のスレッド上でメッセージ・リスナーを作成します。アプリケーションは、キュー・マネージャーがその活動化プロセスを継続できるように、できるだけ早くキュー・マネージャーに制御を戻します。アプリケーションが作成したスレッドはアクティブのままです。

キュー・マネージャーがクローズ要求を受け取るときに立ち上げられるアプリケーションの例

```
public class ExampleCloseApp extends   MQe implements MQeRunListInterface
{
  MQeQueueManager qmgr = null;
  ...
  /* Called by the  queue manager to activate the application  */
  public Object activate( Object owner, Hashtable loadTable,
                          MQeFields setupData )
  {
    qmgr = (MQeQueueManager)owner; /* QMgr is owner of the application */
    performAction(); /* Perform some action */
    /* don't return control to the QMgr until application has finished */
    return (null);
  }
}
```

この例では、キュー・マネージャーが close 要求を受け取るとき、activate() メソッドを使用してアプリケーションが活動化されます。アプリケーションは、その処理を完了するまでキュー・マネージャーに制御を戻してはなりません。なぜなら、キュー・マネージャーが一度制御を取ると、そのクローズ・プロセスを継続するからです。

メッセージ

31ページの『第3章 MQeFields』の説明にもあるとおり、 MQSeries Everyplace メッセージは MQeFields の下位オブジェクトです。アプリケーションは、データを <名前、データ> の対としてメッセージに入れることができます。MQSeries Everyplace は、メッセージング・アプリケーションにとって有用ないくつかの定数フィールド名を定義します。これらのフィールドは、次のとおりです。

固有 ID
: MQe.Msg_OriginQMgr + MQe.Msg_Time

メッセージ ID
: MQe.Msg_ID

相関 ID
: MQe.Msg_CorrelID

優先順位
: MQe.Msg_Priority

固有 ID は、メッセージの作成時にメッセージ・オブジェクトによって生成された固有のタイム・スタンプ (JVM あたり 1 つ) と、メッセージが最初に送られたキュー・マネージャーの名前の組み合わせです。 *固有 ID* は、アプリケーションがメッセージを検索するときに使用します。アプリケーションがこれを変更することはできません。。

MQSeries Everyplace ネットワーク内のすべてのキュー・マネージャーが一意的に命名されている限り、 *固有 ID* を使用して、MQSeries Everyplace ネットワーク内のメッセージを一意的に識別することができます。

注: MQSeries Everyplace がキュー・マネージャー名の固有性を検査したり、それを施行したりすることはありません。キュー・マネージャー名が固有になるよう確認するのは、個々のソリューションの責任です。

getMsgUIDFields() メソッドは、メッセージの*固有 ID* にアクセスします。

```
MQeFields msgUID = msgObj.getMsgUIDFields();
```

getMsgUIDFields() メソッドを使用すると、以下の 2 つのフィールドを持つ MQeFields オブジェクトが返されます。

- MQe.Msg_OriginQMgr
- MQe.Msg_Time

これらのフィールドは、次のようにして個々に検索することができます。

```
long timeStamp    = msgUID.getLong( MQe.Msg_Time );
String originQMgr = msgUID.getAscii( MQe.Msg_OriginQMgr );
```

MQSeries *メッセージ ID* と *相関 ID* フィールドにより、アプリケーションはメッセージの ID を提供することができます。これらの 2 つのフィールドはまた、 MQSeries ファミリーの残りのメンバーとの対話においても使用されます。

```
MQeMsgObject msgObj = new MQeMsgObject;
msgObj.putArrayOfByte( MQe.Msg_ID, MQe.asciiToByte( "1234" ) );
```

*優先順位*フィールドには、メッセージ優先順位値が入ります。メッセージ優先順位は、MQSeries ファミリーの他のメンバーの場合と同様の方法で定義されます。優先順位の範囲は 9 (最高) 〜 0 (最低) です。アプリケーションはこのフィールドを使用して、優先順位に応じて適切にメッセージを処理することができます。

```
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putByte( MQe.Msg_Priority, (byte)8 );
```

アプリケーションは、それぞれ独自のデータ用のフィールドを、メッセージ内に作成することができます。

```
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "PartNo", "Z301" );
msgObj.putAscii( "Colour", "Blue" );
msgObj.putInt( "Size", 350 );
```

代わりの方法として、MQeMsgObject を拡張して、メッセージを作成するのに助けとなるメソッドをいくつか組み込むことができます (以下の例を参照してください)。:

```
package messages.order;
import com.ibm.mqe.*;

/*** This class defines the Order Request format */
public class OrderRequestMsg extends MQeMsgObject
{

  public OrderRequestMsg() throws Exception
  {
  }

 /*** This method sets the client number */
  public void setClientNo(long aClientNo) throws Exception
  {
    putLong("ClientNo", aClientNo);
  }

 /*** This method returns the client number */
  public long getClientNo() throws Exception
  {
    return getLong("ClientNo");
  }

 /*** This method sets the name of the item to be ordered */
  public void setItem(String anItem) throws Exception
  {
    putUnicode("Item", anItem);
  }

 /*** This method returns the name of the item to be ordered */
  public String getItem() throws Exception
  {
    return getUnicode("Item");
  }

 /*** This method sets the quantity required */
  public void setQuantity(int aQuantity) throws Exception
  {
    putInt("Quantity", aQuantity);
```

```
  }

 /*** This method returns the quantity required */
  public int getQuantity() throws Exception
  {
    return getInt("Quantity");
  }

 /*** This method sets the name of the queue to which to send an order reply */
  public void setReplyToQ(String aMyReplyToQ) throws Exception
  {
    putAscii("Msg_ReplyToQ", aMyReplyToQ);
  }

 /*** This method returns the name of the queue to which an order reply will be sent */
  public String getReplyToQ() throws Exception
  {
    return getAscii("Msg_ReplyToQ");
  }

 /*** This method sets the name of the queue manager to which an order reply will be sent */
  public void setReplyToQMgr(String aMyReplyToQMgr) throws Exception
  {
    putAscii("Msg_ReplyToQMgr", aMyReplyToQMgr);
  }

 /*** This method returns the name of the queue manager to which an order reply will be sent */
  public String getReplyToQMgr() throws Exception
  {
    return getAscii("Msg_ReplyToQMgr");
  }
}
```

追加のメソッドは、メッセージ・オブジェクトに出入りするデータの**書き込み**および**取得**を処理します。アプリケーション・プログラマーは、送信されるデータのタイプも、メッセージ内で使用されるフィールド名も知る必要がありません (以下の例を参照してください)。

```
OrderRequestMsg orderRequest = new OrderRequestMsg();
orderRequest.setClientNo( 1234 ); /* client ref. number */
orderRequest.setItem( "  MQSeries Everyplace Programmers Guide" );
/* item being ordered */
orderRequest.setQuantity( 12 ); /* quantity */

/*** send the order reply to QMgr1.OrderReplyQueue */
orderRequest.setReplyToQMgr( "QMgr1" );
orderRequest.setReplyToQ( "OrderReplyQueue" );
```

メッセージの保管

ほとんどのキュー・タイプはメッセージを永続ストアに保持しています。メッセージがストア内にある場合、メッセージの状態は、ストアを出入りするときに変化します。86ページの図9を参照してください。

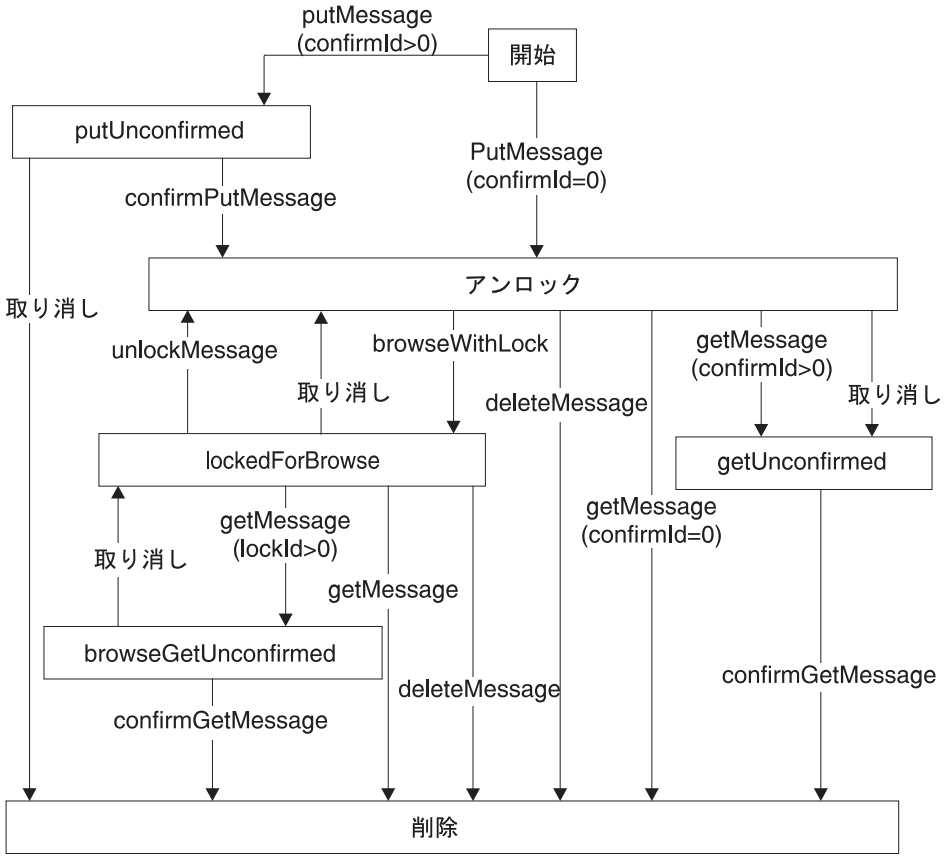

図 9. 保管メッセージ状態の流れ

メッセージ状態

考えられるメッセージ状態は以下のとおりです。

開始 (Start)
メッセージがメッセージ・ストアに追加される前の、メッセージの初期状態。

Put 未確認 (Put Unconfirmed)
あるメッセージが *confirmID* の下のメッセージ・ストアに入れられたが、その追加がまだ確認されていない状態。このメッセージは、**confirmPutMessage**、**confirm**、および **undo** を除くすべてのアクションから、実質上隠されます。

アンロック (Unlocked)
メッセージがメッセージ・ストアに追加されました。このメッセージにはロックがないので、すべての照会で見ることができます。

ブラウズのためのロック (Locked for Browse)
ロック付きブラウズがメッセージを取得しました。このメッセージは、**getMessage**、 **unlockMessage**、および **undo** を除くすべての照会から隠されます。

Get 未確認 (Get Unconfirmed)
Get メッセージが *confirmID* で作成されましたが、その Get が確認されていません。このメッセージは、**confirmGetMessage**、 **confirm**、および **undo** を除くすべての照会に対して非表示になります。これらの各アクションでは、一致する *confirmID* を組み込んで Get を確認する必要があります。

Get 未確認のブラウズ (Browse Get Unconfirmed)
ブラウズがロックされているメッセージが取得されました。この状態は、正しい *lockID* を **getMessage** 機能に渡すことによってのみ起こすことができます。

削除 (Deleted)
メッセージがデータベースから除去された後の最終状態。

メッセージ・イベント

メッセージは、イベントの結果に応じて、状態が変わります。可能なメッセージ・イベントは以下のとおりです (86ページの図9 を参照してください)。

putMessage
メッセージ・ストアに入れられたメッセージで、確認の必要はありません。

getMessage
メッセージ・ストアから検索されたメッセージで、確認の必要はありません。

putMessage with confirmId>0
メッセージ・ストアに入れられたメッセージで、確認の必要があります。

confirmPutMessage
上の **putMessage** with confirmId>0 に対する確認。

getMessage with confirmId>0
メッセージ・ストアから検索されたメッセージで、確認の必要があります。

confirmGetMessage
上の **getMessage** with confirmId>0 に対する確認。

browseWithLock
メッセージをブラウズし、一致するメッセージをロックします。ブラウズ中にメッセージが変わらないようにします。

unlockMessage
ブラウズでロックされたメッセージをアンロックします。

undo ブラウズでロックされたメッセージをアンロックするか、または **getMessage** with confirmId>0 または **putMessage** with confirmId>0 をやり直します。

deleteMessage
メッセージをメッセージ・ストアから除去します。

メッセージ・イベントとメッセージ状態の詳細説明が、98ページの『確実なメッセージ送達』と92ページの『ブラウズおよびロック』にあります。

メッセージ索引フィールド

メモリー・サイズの制約により、完全なメッセージはメモリー内には保持されませんが、メッセージの検索をより速く行うために、MQSeries Everyplace は各メッセージの特定のフィールドをメッセージ*索引* の中に保持します。索引の中に保持されるフィールドは、次のとおりです。

固有 ID
`MQe.Msg_OriginQMgr + MQe.Msg_Time`

メッセージ ID
`MQe.Msg_ID`

相関 ID
`MQe.Msg_CorrelID`

優先順位
`MQe.Msg_Priority`

Status 86ページの『メッセージ状態』に示されている状態のうちの 1 つ。

フィルターでこれらのフィールドを使用すると、 MQSeries Everyplace はすべてのメッセージをメモリーにロードする必要がないため、検索が一層効果的に行われます。

メッセージ・ストアのカスタマイズ

MQSeries Everyplace を使用すれば、ユーザーはキュー・メッセージ・ストア用の独自の特性を定義することができます。クラス MQeAbstractMessageStore をサブクラスに分割して、キュー・ストレージの以下の局面を制御することができます。

- メッセージの保管
- フィルターに掛けたメッセージの検索
- メッセージの暗黙の有効期限切れ
- メッセージの削除
- Put および Get の確認の管理
- メッセージ・ロックの管理
- メッセージ・ブラウズの許可
- メッセージが圧縮 / セキュリティー属性をもっている場合の、効率的かつ安全なメッセージのラッピング

デフォルト・インプリメンテーションでは、以下のメッセージ・フィールドを索引で使用しています。

- MQe.Msg_OriginQMgr
- MQe.Msg_Time
- MQe.Msg_MsgID
- MQe.Msg_CorrelID
- MQe.Msg_Priority
- MQe.Msg_ExpireTime
- MQe.Msg_LockID
- MSG_DESTINATION_QUEUEMANAGER
- MSG_DESTINATION_QUEUE

サブクラスをインプリメントする場合は、索引で使用するフィールドを自由に選択することができます。

サンプル・クラス MQeMessageStore は、このクラスを使用してメッセージ・ストレージをカスタマイズする代表的な例を示したものです。

フィルター

フィルターの概念により、MQSeries Everyplace は強力なメッセージ検索を実行することができます。キュー・マネージャーの主な操作のほとんどは、フィルターの使用をサポートします。フィルターは、MQeFields オブジェクトにフィールドを配置することによって作成できます。たとえば、単純な **get** メッセージ操作で、"ヌル"のフィルターが使用されると、この操作の結果として、キュー上にある使用可能な最初のメッセージが戻されます。

```
qmgr.getMessage( "myQMgr", "myQueue", null, null, 0 );
```

フィルターの使用により、アプリケーションは、フィルターと同じフィールドおよび値を含む、使用可能な最初のメッセージを戻すことができます。たとえば、次のコードにより、メッセージ ID が "1234" の最初のメッセージを取得するフィルターが作成されます。

```
MQeFields filter = new MQeFields();
filter.putArrayOfByte( MQe.Msg_MsgID, MQe.AsciiToByte( "1234" ) );
```

次に、フィルターは、**get** メッセージ操作に渡されます。

```
qmgr.getMessage( "myQMgr", "myQueue", filter, null, 0 );
```

フィルターを検索に適用すると、フィルター内のフィールドが順々に各索引項目と比較されます。あるフィールドが索引項目とフィルターの両方に共通で、そのフィールドの各値が異なる場合は、メッセージがフィルターと一致することはなく、そのメッセージは考慮から排除されます。あるフィールドがフィルターと索引項目の両方に共通でない場合、またはそのフィールドが共通で各値が同じである場合は、メッセージが検索に取り入れられます。

メッセージの有効期限

キューには有効期限を定義することができます。メッセージの保管期間がこの期限を超えた場合には、有効期限切れのマークが付けられます。有効期限切れというマークが付いたときにメッセージがどのように処理されるかは、キュー・ルールによって決まります。

メッセージには、それ自体の有効期限が指定されることもあります。これは、メッセージに MQe.Msg_ExpireTime フィールドを追加することによって定義されます。有効期限には、相対的なもの (例、メッセージの作成後 2 日で満了) もあれば、絶対的なもの (例、2000 年 11 月 25 日 08:00 時) もあります。

下記の例では、メッセージは作成後 60 秒で有効期限が切れます。 (60000 milliseconds = 60 seconds).

```
/* create a new message */
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "MsgData", getMsgData() );
/* expiry time of sixty seconds after message was created */
msgObj.putInt( MQe.Msg_ExpireTime, 60000 );
/* put message onto queue */
qmgr.putMessage( null, "MyQueue", msgObj, null, 0 );
```

下記の例では、メッセージは 2001 年 5 月 15 日 15:25 に有効期限が切れます。

```
/* create a new message */
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "MsgData", getMsgData() );
/* create a Date object for 15th May 2001, 15:25 hours */
Calendar calendar = Calendar.getInstance();
calendar.set( 2001, 04, 15, 15, 25 );
Date expiryTime = calendar.getTime();
/* add expiry time to message */
msgObj.putLong( MQe.Msg_ExpireTime, expiryTime.getTime() );
/* put message onto queue */
qmgr.putMessage( null, "MyQueue", msgObj, null, 0 );
```

キュー

キュー・マネージャーは、メッセージを保持するキューを管理します。キュー・エンティティーは、アプリケーションには直接見えません。キューとの対話はすべてキュー・マネージャーを通して行われます。それぞれのキュー・マネージャーには、それが管理し、所有するキューを持つ機能があります。これらのキューは*ローカル・キュー*と呼ばれます。 MQSeries Everyplace では、アプリケーションは、別のキュー・マネージャーに属するキュー上のメッセージにアクセスすることもできます。これらのキューはリモート・キューと呼ばれます。ローカル・キューでもリモート・キューでも、同じ一連の操作を利用できます。ただし、メッセージ・リスナーの定義は除きます (93ページの『メッセージ・リスナー』を参照)。

キュー上にあるメッセージは、キューの永続ストアで保持されます (85ページの『メッセージの保管』を参照してください)。キューは、キュー・ストア・アダプターを介して永続ストアにアクセスします。 (289ページの『第10章 MQSeries Everyplace アダプター』を参照してください。) アダプターは、MQSeries Everyplace とハードウェア・デバイス (ディスク、ネットワーク、またはデータベースなどのソフトウェア・ストア) との間のインターフェースです。アダプターは、交換可能なコンポーネントとして設計されているので、デバイスとの対話に使用可能なプロトコルを簡単に交換することができます。キューで使用される支援記憶装置は、MQSeries Everyplace 管理メッセージを使って変更することができます。ただし、キューがアクティブになっているか、またはそれにメッセージが含まれている場合には、支援記憶装置の変更は許可されていません。キューで使用される支援記憶装置を使ってシステム障害時にメッセージを回復することができる場合、 MQSeries Everyplace はメッセージの送達を確実に実行することができます。

キュー・タイプ

MQSeries Everyplace キュー・タイプについては、 4ページの『MQSeries Everyplace キュー』で簡潔に説明しています。さまざまなタイプのセットアップと管理については、139ページの『キュー』を参照してください。

キューの順序付け

キュー上のメッセージの順序は、基本的に優先順位によって決まります。メッセージ優先順位の範囲は 9 (最高) 〜 0 (最低) です。優先順位値が同じメッセージはキューに到達した時刻によって配列されます。したがって、最も長時間キュー上にあるメッセージは、その優先順位グループの先頭に配置されます。

キュー上のすべてのメッセージの読み取り

キューが空のとき、**get** message コマンドが実行された場合、キューは `Except_Q_NoMatchingMsg` 例外を出します。このことにより、キュー上のすべてのメッセージを読み取るアプリケーションを作成することができます。

getMessage() 呼び出しを `try..catch` ブロック内部に入れることにより、発生する例外コードをテストすることができます。これは、MQeException クラスの **code()** メソッドを使用して行われます。 **code()** メソッドの結果と、 MQe クラスによって発行される例外定数のリストを比較することができます。例外のタイプが `Except_Q_NoMatchingMsg` でない場合は、もう一度例外を出します。

次のコードはこの技法を示しています。

```
try {
  while( true )
  { /* keep getting messages until an exception is thrown */
    MQeMsgObject msg = qmgr.getMessage( "myQMgr", "myQueue", null, null, 0 );
    processMessage( msg );
  }
```

```
}
catch ( Exception e )
{
  if ( e.code() != MQe.Except_Q_NoMatchingMsg )
        throw e;
}
```

ブラウズおよびロック

メッセージのグループのブラウズし、それらをロックすることによって、アプリケーションは、メッセージがロックされている間、他のアプリケーションによってメッセージが処理されないようにすることができます。アプリケーションによってアンロックされるまで、メッセージはロックされたままの状態になります。他のアプリケーションがメッセージをアンロックすることはできません。

```
MQeEnumeration msgEnum = qmgr.browseMessagesAndLock( null, "MyQueue", null,
                                                     null, 0, false );
```

このコマンドは、ローカル・キュー・マネージャー上に存在する MyQueue キュー上にあるすべてのメッセージをロックします (ヌルは、ローカル・キュー・マネージャーの別名です)。現在、これらのメッセージには、それらをロックしたアプリケーションだけがアクセスすることができます。 (**ブラウズおよびロック**操作の後にキューに届いたメッセージは、ロックされません。)

MQeEnumeration オブジェクトには、ブラウズに提供されたフィルターに一致するすべてのメッセージが入ります。 MQeEnumeration は、標準 Java Enumeration と同じ仕方で使用することができます。次のようにして、ブラウズされたすべてのメッセージを列挙することができます。

```
while( msgEnum.hasMoreElements() )
{
  MQeMsgObject msg = (MQeMsgObject)msgEnum.nextElement();
  System.out.println( "Message from  queue manager: " +
                      msg.getAscii( MQe.Msg_OriginQMgr ) );
}
```

アプリケーションは、メッセージに対して **get** または **delete** のいずれかの操作を実行して、それらをキューから除去することができます。これを行うには、アプリケーションは列挙したメッセージとともに戻される*ロック ID* を提供する必要があります。 *lock ID* を指定することにより、最初にそれらをアンロックしなくても、アプリケーションはロックされたメッセージを処理することができます。次のコードは、列挙で戻されるすべてのメッセージに対して、 **delete** を実行します。メッセージの*固有 ID* と*ロック ID* は、 **delete** 操作でフィルターとして使用されます。

```
while( msgEnum.hasMoreElements() )
{
  MQeMsgObject msg = (MQeMsgObject)msgEnum.getNextMessage( null,0 );

  processMessage( msg );
  MQeFields filter = msg.getMsgUIDFields();
```

```
  filter.putLong( MQe.Msg_LockID, msgEnum.getLockId() );

  qmgr.deleteMessage( null, "MyQueue", filter );
}
```

標準の **java.util.Enumeration nextElement()** メソッドを使用する代わりに、MQeEnumeration は **getNextMessage()** メソッドを提供します。このメソッドは、**browseMessages()** メソッドの *justUID* パラメーターに応じた仕方で作動します。このパラメーターは、ブラウズ操作がメッセージ内の一致するすべてのフィールドを戻すか、あるいは*固有 ID* フィールドだけを戻すかを指定します。

justUID パラメーターが `false` に設定されている場合、ブラウズによって戻される MQeEnumeration には、一致するメッセージのすべてのフィールドが含まれることになります。この場合、**getNextMessage()** メソッドは、 **nextElement()** と同様に作動します。

justUID パラメーターが `true` に設定されている場合、ブラウズによって戻される MQeEnumeration には、一致するメッセージの*固有 ID* フィールド (`MQe.Msg_OriginQMgr` および `MQe.Msg_TimeStamp`) だけが含まれます。この場合、適切な **get** メッセージが実行され、そのメッセージはキューから除去されます。

メッセージの取得の際には確実なメッセージ送達を用いることができます。ゼロ以外の*確認 ID* を指定するとは、取得の確認が必要であるという意味です (確実なメッセージ送達の詳細については、98ページの『確実なメッセージ送達』を参照してください)。

メッセージをキューから除去する代わりに、メッセージをアンロックすることもできます。これで、メッセージが再びすべての MQSeries Everyplace アプリケーションに見えるようになります。このことは、**unlockMessage()** メソッドを使用して行うことができます。

注: MQSeries-ブリッジ キューに関する特殊考慮事項については、206ページの『MQSeries-ブリッジ キューからのメッセージの取得およびブラウズ』を参照してください。

メッセージ・リスナー

MQSeries Everyplace では、アプリケーションはキュー上で発生したイベントを *listen* することができます。通知は標準 Java イベントの形式を取り、 listen しているアプリケーションは、イベントの発生時に呼び出されるメソッドを提供するインターフェースをインプリメントします。アプリケーションは、関係のあるメッセージを識別するためのメッセージ・フィルターを指定することができます。

```
/* Create a filter for "Order" messages of priority 7 */
MQeFields filter = new MQeFields();
filter.putAscii( "MsgType", "Order" );
```

```
filter.putByte( MQe.Msg_Priority, (byte)7 );
/* activate a listener on "MyQueue" */
qmgr.addMessageListener( this, "MyQueue", filter );
```

addMessageListener() メソッドに渡されるパラメーターは、次のとおりです。

- メッセージ・イベントを listen するキューの名前。
- MQeMessageListenerInterface をインプリメントするコールバック・オブジェクト。
- メッセージ・フィルターを含む MQeFields オブジェクト。

メッセージがリスナーの付加されたキューに達すると、キュー・マネージャーは、メッセージ・リスナーの作成時に与えられた *callback* オブジェクトを呼び出します。

以下の例は、アプリケーションがメッセージ・イベントを処理する通常の方法を示しています。

```
public void messageArrived(MQeMessageEvent msgEvent )
{
String queueName =msgEvent.getQueueName();
if (queueName.equals("MyQueue"))
{
       try
       {
/*get message from queue */
MQeMsgObject msg =qmgr.getMessage(null,queueName,
msgEvent.getMsgFields(),null,0 );

processMessage(msg );
       }
       catch (MQeException  e)
      {
      ...
      }
}
}
```

messageArrived() は、MQeMessageListenerInterface にインプリメントされたメソッドの 1 つです。 *msgEvent* パラメーターにはメッセージに関する情報が含まれています。これには次のものが含まれます。

- メッセージが達するキューの名前
- メッセージの *UID*
- メッセージ *ID*
- *相関 ID*
- メッセージ*優先順位*

メッセージ・フィルターは、ローカル・キュー上でのみ作動します。ポーリングと呼ばれる別の技法により、メッセージがリモート・キューに達するとすぐにそれを取得することができます。

メッセージ・ポーリング

メッセージ・ポーリングは **waitForMessage()** メソッドを使用します。このコマンドは、リモート・キューに対して定期的に **getMessage()** コマンドを実行します。指定されたフィルターに一致するメッセージが使用可能になるとすぐに、呼び出し側アプリケーションにそのメッセージが戻されます。

次は、メッセージ呼び出し待機の典型的な例です。

```
qmgr.waitForMessage( "RemoteQMgr", "RemoteQueue", filter, null, 0, 60000 );
```

waitForMessage() メソッドは、最後のパラメーターで指定された時間だけリモート・キューをポーリングします。この時間はミリ秒単位で指定されます。それで上記の例では、ポーリングは 60 秒間続きます。コマンドが実行されているスレッドは、指定された時間より前にメッセージが戻らない限り、その時間中はブロックされます。

メッセージ・ポーリングは、ローカル・キューとリモート・キューの両方の操作に影響します。

注: この技法を使用すると、多重要求がネットワーク上で送信されることになります。

メッセージング操作

表2 は、さまざまなタイプのキュー上にあるメッセージに対して実行できる操作を示しています。

表 2. メッセージ操作

操作	ローカル・キュー	リモート・キュー	
		同期	非同期
ブラウズ (およびロック)	はい	はい	
delete	はい	はい	
get	はい	はい	
listen	はい		
書き込み	はい	はい	はい
待機	はい	はい	

同期および非同期のメッセージング

MQSeries Everyplace により、アプリケーションはメッセージを柔軟に処理することができます。メッセージは、*同期的*にも*非同期的*にも伝送できます。

同期メッセージング

アプリケーションは、そのメッセージがいつどのようにして送信されるかを知る必要はありません。しかし望むならば、同期メッセージングを使用して、このプロセスを制御することができます。同期メッセージングは、**put** message コマンドが実行されるとすぐに、メッセージが送信されるということです。このタイプのメッセージングは、ローカルと宛先の両方のキュー・マネージャーが同時にオンラインであるときにのみ行われます。キュー・マネージャーがネットワークに接続されていない場合には作動しません。同期メッセージは、インスタント接続のパフォーマンス上の利点を提供し、メッセージがその宛先に到達したことを認識します。

非同期メッセージング

非同期メッセージングでは、デバイスがネットワークに接続されているかどうかにかかわりなく、アプリケーションはメッセージの処理を継続することができます。アプリケーションがメッセージをリモート・キュー定義に書き込むと、そのメッセージはキュー・マネージャーによって保管されます。このメッセージは、後で接続が確立されたときにリモート・キュー・マネージャーに送信されます。アプリケーションは、伝送がいつ行われるかを意識する必要はありません。

非同期メッセージングの典型的な例が IBM 技術員または販売員用のアプリケーションです。技術員は、都合の良い時に注文や在庫を送信することができます。デバイスが物理的にネットワークに接続できるまで、メッセージはローカルに保管されます。接続が行われると、そのメッセージの伝送が可能になります。

非同期伝送を行うには、キュー・マネージャーを*起動*する必要があります。起動は、アプリケーションがキュー・マネージャーの **triggerTransmission()** メソッドを呼び出すことによって、あるいはキュー・マネージャーの伝送ルールを使用することによって、行われます (109ページの『伝送ルール』を参照)。 メッセージ伝送のメソッドは、リモート・キューがどのように定義されているかによって決まります。メッセージをリモート・キューに送信するキュー・マネージャーは、そのキューの定義を保持します。この定義は、リモート・キュー定義と呼ばれます。メッセージがリモート・キューに書き込まれると、ローカル・キュー・マネージャーはリモート・キュー定義を使用して、メッセージの送信方法を判別します。

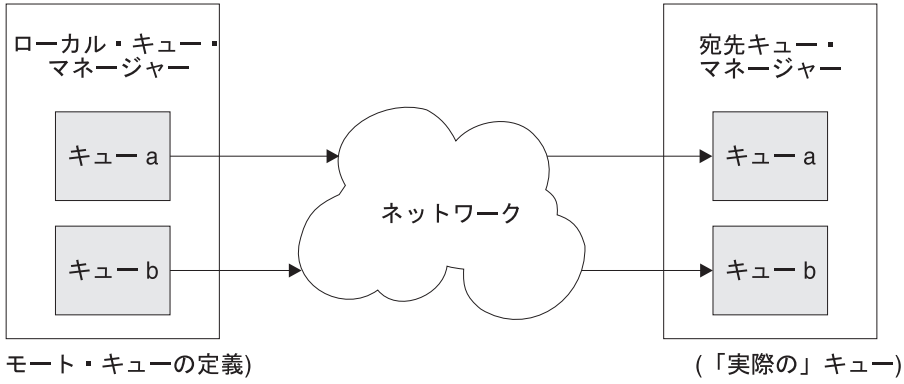

図 10. MQSeries Everyplace メッセージ・フロー

メッセージはローカル・キュー・マネージャーからリモート・キュー・マネージャーに、リモート・キュー上で定義された認証プログラム、暗号機能、および圧縮機能を使用して送信されます。 2 つのキュー・マネージャーの間でメッセージ・チャネルを作成できるようにするためには、ローカル・キュー・マネージャーは、リモート・キュー属性を知っている必要があります。ローカル・キュー・マネージャーは、この情報をそのリモート・キュー定義の一部として保管します。

2 つの伝送スタイルは、これを別々の方法で処理します。

アプリケーションがリモート・キューにメッセージを書き込む際、リモート・キューの定義がローカルに保持されていると、キューの特性はそのリモート・キュー定義を使って判別されます。定義がローカルに保持されていない場合は、キュー・ディスカバリーが行われます。このローカル・キュー・マネージャーは、同期的にリモート・キュー・マネージャーと交信し、キューの特性の確認を試みます。これによって次のような特性が検出されます。

- Queue_Description
- Queue_Expiry
- Queue_MaxQSize
- Queue_MaxMsgSize
- Queue_Priority
- Queue_Cryptor
- Queue_Authenticator
- Queue_Compressor
- Queue_TargetRegistry
- Queue_AttrRule

キューのディスカバリーが正常に行われた後、このキューの定義は、ディスカバリーが開始されたキュー・マネージャーにリモート・キュー定義として保管されます。このようにして検出されたキューの定義は、通常のリモート・キュー定義と同じように扱われます。なお、*Queue_Mode* は、検出されたキューがすべて同期操作のために設定されている場合には検出されません。

しかし、非同期スタイルの伝送は、宛先キュー・マネージャーから情報を要求することができません。したがって、非同期伝送が可能になる前に、リモート・キュー定義が存在していなければなりません。リモート・キュー定義は、MQSeries Everyplace 管理メッセージを使用して作成することができます (119ページの『第6章 メッセージング・リソースの管理』を参照)。

同期メッセージングと非同期メッセージングを組み合わせることによって、MQSeries Everyplace は信頼できない通信リンクに対処することができます。リンクが切断されたためにメッセージをすぐに送達できない場合には、そのメッセージはキューに入れられ、後ほど送達されます。この例を以下に示します。 2 つのキューを定義することによって、アプリケーションは同期伝送を行えない状態に対処することができます。

```
try {
  qmgr.putMessage( "RemoteQMgr", "TransactionQueue", msgObj, null, 0 );
}
catch ( Exception e )
/* reset message UID */
msgObj.resetMsgUIDFields();
{ /* if connection cannot be made, put message on asynchronous queue */
  if ( e.getMessage().equals( "Connection Refused" )
    qmgr.putMessage( "RemoteQMgr", "AsynchTransactionQueue",
                     msgObj, null, 0 );
}
```

確実なメッセージ送達

非同期伝送は、*確実*なメッセージ送達の概念を紹介します。メッセージを非同期に送達すると、MQSeries Everyplace はメッセージを一回限り、その宛先キューに確実に送達します。しかし、この保証は、リモート・キューとリモート・キュー・マネージャーの定義が、リモート・キューとリモート・キュー・マネージャーの現行の特性と一致する場合にのみ有効です。リモート・キュー定義とリモート・キューが一致しない場合は、メッセージが配信不能となる可能性もあります。この場合、メッセージは失われませんが、ローカル・キュー・マネージャーに保管されたままになります。

同期の確実なメッセージ送達

メッセージの書き込み

同期メッセージ伝送を使用して確実なメッセージ送達を実行することができますが、エラー処理については、アプリケーションが実行する責任があります。

メッセージの確実ではない送達は、単一のネットワーク・フローで生じます。メッセージを送信するキュー・マネージャーは、宛先キュー・マネージャーへのチャネルを作成し、そのチャネルにトランスポーターを接続します。トランスポーターは宛先キューを指します。 (適切なチャネルおよびトランスポーターは、以前の操作から存在していることがあります。その場合には、代わりに既存のものが使用されます。)

送信されるメッセージはダンプされてバイト・ストリームを作成します。このバイト・ストリームは伝送用のチャネルに送られます。プログラム制御が一度チャネルから戻されると、送信側キュー・マネージャーは、メッセージが正常に宛先キュー・マネージャーに送られ、その宛先がメッセージのログをキュー上に記録したこと、およびメッセージが MQSeries Everyplace アプリケーションに見えるようになっていることを認識します。

しかし、送信側が宛先からチャネルを介して例外を受け取る場合、問題が生じることがあります。送信側は、例外の発生が、メッセージがログに記録されて見えるようになる前なのか、それともその後なのかを知る方法がありません。例外が発生したのがメッセージが見えるようになる前であった場合は、送信側が安全にメッセージを再送することができます。しかし、それがメッセージが見えるようになった後の場合には、メッセージを重複してシステムに送信する恐れがあります。なぜなら、MQSeries Everyplace アプリケーションは、送信側が 2 度目にメッセージを送信する前に、メッセージを処理してしまった可能性があるからです。

この問題の解決策には、追加の確認フローを伝送することが関係しています。送信側アプリケーションがこのフローが成功したという応答を受け取るならば、メッセージが一回だけ送達されたことが分かります。

putMessage メソッドの *confirmId* パラメーターは、確認フローを送信するかどうかを示します。 **ゼロ**の値は、メッセージ伝送が 1 つのフローで生じることを意味し、 **ゼロ以外**の値は、確認フローが予期されていることを意味します。宛先キュー・マネージャーは、通常どおりメッセージを宛先キューに記録しますが、確認フローを受け取るまでは、メッセージはロックされ、MQSeries Everyplace アプリケーションには見えません。

MQSeries Everyplace アプリケーションは、 **confirmPutMessage** メソッドを使用して、**put** message confirmation を発行することができます。宛先キュー・マネージャーはこのコマンドによって生成されるフローを受け取ると、メッセージをアンロックし、それを MQSeries Everyplace アプリケーションに見えるようにします。ただし、一度に 1 つのメッセージだけしか確認することができず、メッセージのバッチを確認することはできません。

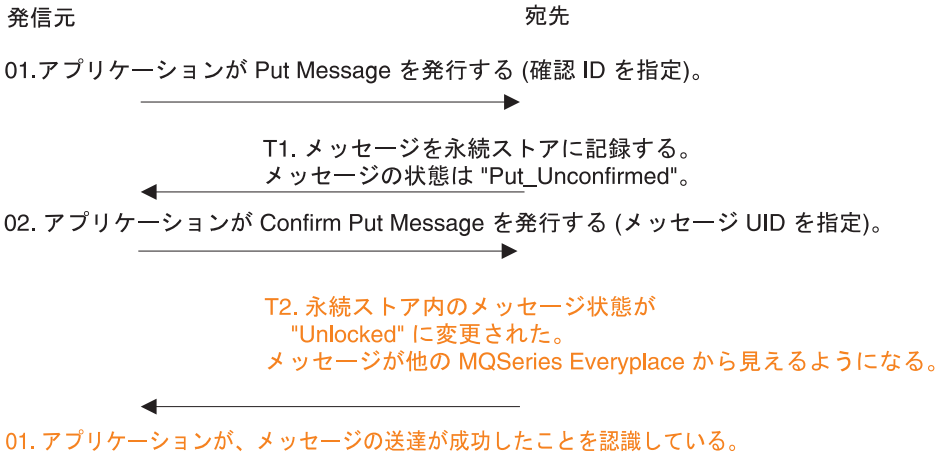

図 11. 同期メッセージの確実な書き込み

confirmPutMessage() メソッドでは、前の put message コマンドで使用された *Confirm ID* ではなく、 メッセージの *UID* を指定する必要があります。 (*Confirm ID* も、伝送失敗の後ロックされたままになっているメッセージを復元するために使用されます。これについては、103ページで詳しく説明しています。)

確実な**書き込み**のために必要なコードの骨組みを示します。

```
long confirmId = MQe.uniqueValue();
try {
    qmgr.putMessage( "RemoteQMgr", "RemoteQueue", msg, null, confirmId );
}
catch( Exception e )
{
    /* handle any exceptions */
}
try {
  qmgr.confirmPutMessage( "RemoteQMgr", "RemoteQueue",
                          msg.getMsgUIDFields() );
}
catch ( Exception e )
{
    /* handle any exceptions */
}
```

図11 のステップ 1 で障害が発生した場合、アプリケーションはメッセージを再送する必要があります。この場合、重複したメッセージが MQSeries Everyplace ネットワークに送信される恐れはありません。なぜなら、宛先キュー・マネージャーのメッセージは、確認フローが処理されるまでは、アプリケーションには見えないからです。

MQSeries Everyplace アプリケーションがメッセージを再送する場合、それを行うことを宛先キュー・マネージャーに通知する必要もあります。宛先キュー・マネージャーは、

すでに持っているメッセージの重複コピーを削除します。アプリケーションはこれを行うために MQe.Msg_Resend フィールドを設定します。

100ページの図11のステップ 2 で障害が発生した場合、アプリケーションはもう一度確認フローを送信する必要があります。このとき、宛先キュー・マネージャーは、すでに確認したメッセージについて受け取った確認フローは無視するため、安心して再送することができます。

以下のコードは、examples.application.example6 からの抜粋です。

```
boolean msgPut     = false; /* put successful? */
boolean msgConfirm = false; /* confirm successful? */
int maxRetry       = 5; /* maximum number of retries */

long confirmId = MQe.uniqueValue();

int retry = 0;
while( !msgPut && retry < maxRetry )
{
  try
  {
    qmgr.putMessage( "RemoteQMgr", "RemoteQueue", msg, null, confirmId );
   msgPut = true;  /* message put successful             */
  }
  catch( Exception e )
  {
    /* handle any exceptions */
    /* set resend flag for retransmission of message */
    msg.putBoolean( MQe.Msg_Resend, true );
    retry ++;
  }
}

if ( !msgPut )  /* was put message successful?   */
    /* Number of retries has exceeded the maximum allowed, so abort the put*/
    /* message attempt                                                     */
  return;

retry = 0;
while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmPutMessage( "RenoteQMgr", "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true;   /* message confirm successful        */
  }
  catch ( Exception e )
  {
    /* handle any exceptions */
    /* An Except_NotFound exception means that the message has already   */
    /* been confirmed                                                    */
    if ( e instanceof MQeException &&
         ((MQeException)e).code() == Except_NotFound )
```

```
      putConfirmed = true;            /* confirm successful            */
    /* another type of exception - need to reconfirm message           */
    retry ++;
  }
}
```

メッセージの取得

確実なメッセージの**取得**は、**書き込み**と同様の方法で行われます。メッセージの**取得**コマンドが、ゼロより大きい *confirmId* パラメーターを指定して実行されると、確認フローが宛先キュー・マネージャーによって処理されるまで、メッセージはそれが存在するキュー上でロックされたままになります。確認フローが受信されると、そのメッセージはキューから削除されます。

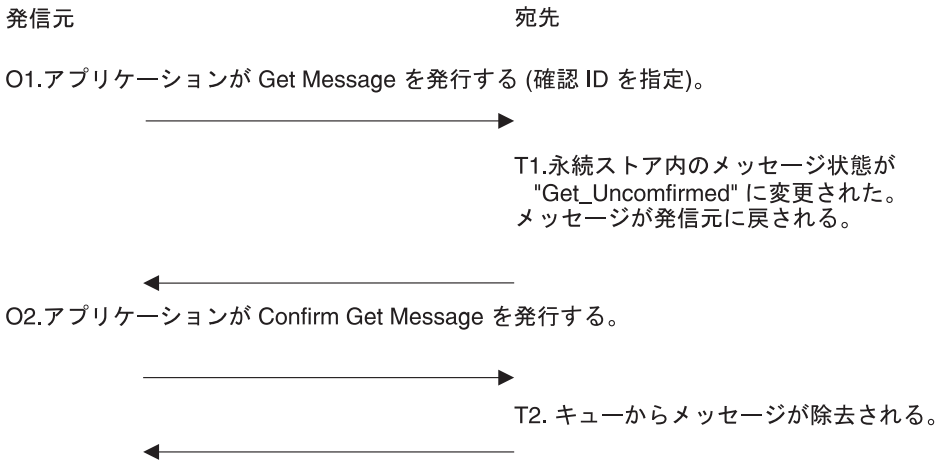

図 12. 同期メッセージの確実な取得

以下のコードは、examples.application.example6 からの抜粋です。

```
boolean msgGet     = false; /* get successful? */
boolean msgConfirm = false; /* confirm successful? */
MQeMsgObject msg   = null;
int maxRetry       = 5; /* maximum number of retries */
long confirmId = MQe.uniqueValue();
int retry = 0;
while( !msgGet && retry < maxRetry )
{
  try
  {
    msg = qmgr.getMessage( "RemoteQMgr", "RemoteQueue", filter, null,
                           confirmId );
    msgGet = true; /* get succeeded */
  }
  catch ( Exception e )
```

```
  {
    /* handle any exceptions  */
    /* if the exception is of type Except_Q_NoMatchingMsg, meaning that  */
    /* the message is unavailable then throw the exception               */
    if ( e instanceof MQeException )
      if ( ((MQeException)e).code()  == Except_Q_NoMatchingMsg )
        throw e;
    retry ++; /* increment retry count */
  }
}

if ( !msgGet )                          /* was the get successful?       */
    /* Number of retry attempts has exceeded the maximum allowed, so abort  */
    /* get message operation                                                */
  return;

while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmGetMessage( "RemoteQMgr", "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true; /* confirm succeeded */
  }
  catch ( Exception e )
  {
    /* handle any exceptions  */
    retry ++; /* increment retry count */
  }
}
```

confirmId パラメーターとして渡される値には、別の使用法もあります。この値は、ロックされて確認を待っている間、メッセージを識別するために使用されます。 **get** 操作中にエラーが発生すると、キュー上でメッセージがロックされたままになることがあります。 **get** コマンドの応答としてメッセージがロックされ、アプリケーションがメッセージを受け取る前にエラーが発生すると、このようなことが起こります。アプリケーションが例外の応答として **get** を再発行した場合には、メッセージがロックされて MQSeries Everyplace アプリケーションに見えなくなるため、同じメッセージを受け取ることができなくなります。

しかし、**get** コマンドを実行したアプリケーションは、 **undo** メソッドを使用して、メッセージを復元することができます。アプリケーションは、メッセージの**取得**コマンドに提供した *confirmId* 値を提供する必要があります。 **undo** コマンドは、メッセージを **get** コマンドの前の状態に復元します。

```
boolean msgGet     = false; /* get successful? */
boolean msgConfirm = false; /* confirm successful? */
MQeMsgObject msg   = null;
int maxRetry       = 5; /* maximum number of retries */

long confirmId = MQe.uniqueValue();
int retry = 0;
```

```
while( !msgGet && retry < maxRetry )
{
  try
  {
    msg = qmgr.getMessage( "RemoteQMgr", "RemoteQueue", filter, null,
                           confirmId );
    msgGet = true; /* get succeeded */
  }
  catch ( Exception e )
  {
    /* handle any exceptions  */
    /* if the exception is of type Except_Q_NoMatchingMsg, meaning that  */
    /* the message is unavailable then throw the exception               */
    if ( e instanceof MQeException )
      if ( ((MQeException)e).code()  == Except_Q_NoMatchingMsg )
        throw e;
    retry ++; /* increment retry count */
    /* As a precaution, undo the message on the queue. This will remove  */
    /* any lock that may have been put on the message prior to the       */
    /* exception occurring                                               */
    myQM.undo( qMgrName, queueName, confirmId );
  }
}

if ( !msgGet )                          /* was the get successful?       */
    /* Number of retry attempts has exceeded the maximum allowed, so abort  */
    /* get message operation                                                */
  return;

while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmGetMessage( "RemoteQMgr", "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true; /* confirm succeeded */
  }
  catch ( Exception e )
  {
    /* handle any exceptions  */
    retry ++; /* increment retry count */
  }
}
```

undo コマンドも、 **putMessage** および **browseMessagesAndLock** コマンドと関係があります。メッセージの**取得**の場合と同様に、**undo** コマンドは **browseMessagesandLock** コマンドによってロックされたメッセージを、以前の状態に復元します。

putMessage コマンドが失敗した後でアプリケーションが **undo** コマンドを実行すると、宛先キュー上でロックされて確認を待っているメッセージが削除されます。

undo コマンドは、ローカル・キューとリモート・キューの両方の操作に影響します。

セキュリティー

キュー・マネージャーは、MQSeries Everyplace に提供されるセキュリティー機能を完全にサポートします。セキュリティー特性を使用して定義されたキューに保管されているメッセージは、それらの特性を使ってエンコードされます。キュー・マネージャーとセキュア・キューとの間でセットアップされた通信チャネルは、キューのセキュリティー特性を使用します。または、同等かそれ以上のセキュリティーを持つ既存のチャネルを使用することもあります。

セキュリティー特性をメッセージに直接付加することにより、メッセージを個々に保護することができます。この方法で保護されたメッセージを扱うときには必ず、正確な特性が存在していなければなりません。

MQSeries Everyplace セキュリティーの詳細については、221ページの『第8章 セキュリティー』を参照してください。

第5章 ルール

MQSeries Everyplace は、その主なコンポーネントの動作を制御するために、*rules* の概念を使用します。ルールを使用すると、ソリューションは MQSeries Everyplace の内部動作をある程度制御することができます。ルールは、MQSeries Everyplace コンポーネントが初期化されたときに、それらによってロードされる Java クラスの形式を取ります。

コンポーネントのルールは、コンポーネントの実行段階の特定の時点で呼び出されます。コンポーネントは、特定の署名のメソッドが使用可能になっていることを予期するため、基本ルールを拡張する際に、正確なメソッド署名を使用するように注意してください。

デフォルトまたは例のルールがすべての MQSeries Everyplace コンポーネントに提供されていますが、ソリューションの要件に適合するように、ソリューションは MQSeries Everyplace の動作をカスタマイズする独自のルールを提供することが期待されています。

キュー・マネージャー・ルール

キュー・マネージャー・ルールが使用されるのは、次の場合です。

- キュー・マネージャーが活動化された。
- キュー・マネージャーがクローズされた。
- キューがキュー・マネージャーに追加された。
- キューがキュー・マネージャーから除去された。
- メッセージの書き込み操作が発生した。
- メッセージの取得操作が発生した。
- メッセージの削除操作が発生した。
- メッセージの取り消し操作が発生した。
- キュー・マネージャーが保留メッセージをすべて送信するよう起動された (送信ルール)。
- 着信対等接続が確立された。

キュー・マネージャー・ルールのロードと活動化

キュー・マネージャー・ルールは、キュー・マネージャー管理メッセージ (キュー・マネージャー・ルール・クラス更新の要求が含まれている) を受信するたびにロード、または変更されます。

キュー・マネージャー・ルールがすでにキュー・マネージャーに適用されている場合は、現行のルールを別のルールと置き換えることができるかどうかを聞かれます。置き換えることができる場合は、新規のルールがロードされ、活動化されます。 (キュー・マネージャーの再始動は不要です。)

パッケージ examples.administration.commandline に含まれている QueueManagerUpdater コマンド行ツールは、このような管理メッセージを作成する方法を示しています。

キュー・マネージャー・ルールの使用

以下に示すのは、キュー・マネージャー・ルールの使用法のいくつかの例です。

最初の例は、**メッセージの書き込み**ルールを示しています。ここでは、このキュー・マネージャーを使用してキューに書き込まれるすべてのメッセージには、MQSeries Everyplace *メッセージ ID* フィールドが含まれていなければならないことが強調されています。

```
/* Only allow msgs containing an ID field to be placed on the Queue */
public void putMessage( String destQMgr, String destQ, MQeMsgObject msg,
                        MQeAttribute attribute, long confirmId )
{
  if ( !(msg.Contains( MQe.Msg_MsgId )) )
    throw new MQeException( Except_Rule, "Msg must contain an ID" );
}
```

次のルールの例は、**メッセージの取得**ルールです。ここでは、OutboundQueue と呼ばれるキュー上でメッセージの取得要求を処理できるようにするために、前もってパスワードを提供しなければならないことが強調されています。パスワードは、**getMessage()** メソッドに渡されるメッセージ・フィルターに、フィールドとして組み込まれます。

```
/* This rule only allows GETs from 'OutboundQueue', if a password is  */
/* supplied as part of the filter                                     */
public void getMessage( String destQMgr, String destQ, MQeFields filter,
                        MQeAttribute attr, long confirmId )
{
  super.getMessage( destQMgr, destQ, filter, attr, confirmId );
  if ( destQMgr.equals( Owner.GetName() && destQ.equals( "OutboundQueue" ) )
  {
    if ( !(filter.Contains( "Password" ) )
      throw new MQeException( Except_Rule, "Password not supplied" );
    else
    {
      String pwd = filter.getAscii( "Password" );
      if ( !(pwd.equals( "1234" )) )
        throw new MQeException( Except_Rule, "Incorrect password" );
    }
  }
}
```

上記のルールは、キューを保護する単純な例です。さらにセキュリティーを拡張するために、*認証プログラム*を使用することをお勧めします。これにより、アプリケーションはアクセス制御リストを作成し、キューからメッセージを取得できる人物を管理することができます。

次のルールの例は、キュー・マネージャー管理要求がキューを除去しようとしたときに呼び出されます。ルールには、問題のキューへのオブジェクト参照が渡されます。次の例では、ルールは渡されるキューの名前をチェックし、キューが PayrollQueue という名前の場合には、キューの除去要求が拒絶されます。

```
/* This rule prevents the removal of the Payroll Queue */
public void removeQueue( MQeQueue queue ) throws Exception
{
  if ( queue.getQueueName().equals( "PayrollQueue" ) )
    throw new MQeException( Except_Rule, "Can't delete this queue" );
}
```

キュー・マネージャーは、それ独自の対等チャネル・リスナーを定義することができます。リスナーは、対等チャネルを通して行われた他のキュー・マネージャーからの着信接続試行を検出します。次のルールは、接続要求が検出されると呼び出されます。ルールには、接続を試行しているキュー・マネージャーの名前が渡されます。

```
public void peerConnection( String qmgrName )
{
  /* block any connection attempt from 'RogueQMgr' */
  if ( qmgrName.equals( "RogueQMgr" ) )
    throw new MQeException( Except_Rule, "Connection not allowed" );
}
```

伝送ルール

リモート・キューに書き込まれ、同期として定義されるメッセージは、即時に伝送されます。非同期として定義され、リモート・キューに書き込まれるメッセージは、キュー・マネージャーが伝送に起動されるときまで、ローカル・キュー・マネージャーに保管されます。キュー・マネージャーはアプリケーションによって直接起動することができますが、処理はキュー・マネージャーの伝送ルールによっても制御できます。

伝送ルールは、キュー・マネージャー・ルールのサブセットです。

ルール・クラス内には、メッセージ伝送を制御するための次の 2 つのメソッドがあります。

triggerTransmission()
: ルールが呼び出されるときにメッセージ伝送を許可するかどうかを指定します。

transmit()
: それぞれのキューごとに、伝送を許可するかどうかを決定します。たとえば、

優先順位が高いと見なされるキューからのメッセージだけを伝送できるようにします。 **transmit()** ルールは、**triggerTransmission()** ルールが正常に戻る場合にのみ呼び出されます。

トリガー伝送ルール

メッセージがリモート非同期キューに書き込まれると、 MQSeries Everyplace は**トリガー伝送ルール**を呼び出します。キュー・マネージャーの **triggerTransmission** メソッドは、このルールを変更し、保留メッセージの伝送を試行します。

```
/* default transmission rule - always allow transmission */
public boolean triggerTransmission( int noOfMsgs, MQeFields msgFields )
{
  return true;
}
```

このルールからの戻りコードは、保留メッセージを伝送するかどうかをキュー・マネージャーに知らせます。戻りコードが `true` であれば"伝送し"、`false` であれば、この時点では "伝送しません"。したがって、上記のルールでは、すべてのメッセージを即時に伝送しようとします。これは、基本キュー・マネージャー・ルール class com.ibm.mqe.MQeQueueManagerRule に含まれている、デフォルトの **triggerTransmission()** ルールです。ルールはメッセージがキューに書き込まれるとすぐにメッセージを伝送しようとします。この同期モードに近い操作は、すべてのメッセージを別々に送信するので、非効率的です。通常は、メッセージはグループとして送信し、ネットワークをより効率的に使用するほうが勝っています。

もっと複雑なルールでは、メッセージの優先順位に基づいて、即時に伝送するかどうかを指定することができます。次の例は、優先順位が 5 より大きいメッセージが到着した場合に、キュー・マネージャーを起動するルールを示しています。

```
/* Decide to transmit based on priority of message */
public boolean triggerTransmission( int noOfMsgs, MQeFields msgFields )
{
  if ( msgFields == null ) /* msg fields may be null */
    return false;
  if ( !(msgFields.contains( MQe.Msg_Priority )) )
    return false; /* no priority field in message */
  byte priority = msg.GetByte( MQe.Msg_Priority );
  if ( priority > 5 ) /* if message priority greater than 5 */
    return true; /* then transmit */
  else
    return false; /* else do not transmit */
}
```

msgFields パラメーターには、メッセージから選択したフィールドが含まれます。これらのフィールドは、次のとおりです。

- *固有 ID*
- *メッセージ ID*
- *相関 ID*

- *優先順位*

ルールが伝送を許可することを決定した場合、非同期リモート・キューに書き込まれたメッセージだけでなく、すべての保留メッセージが伝送されます。

noOfMsgs パラメーターには、伝送を待機しているメッセージの数が含まれています。ソリューションは、保留メッセージが一定の数になるまで、伝送をブロックするルールをインプリメントするよう指定することができます。このようなルールは、ネットワーク接続をより効率的に使用するのに役立ちます。

以下のルールは、伝送を待機しているメッセージが最低 10 個になるまで、伝送をブロックします。

```
public void triggerTransmission( int noOfMsgs, MQeFields msgFields )
{
  if ( noOfMsgs >= 10 ) /* if more than 10 msgs are waiting */
    return true; /* then transmit */
  else
    return false;
}
```

伝送ルール

transmit() ルールは、**triggerTransmission()** ルールが伝送を許可する場合にのみ呼び出されます (戻り値 true)。 **transmit()** ルールは、伝送を待機しているメッセージを保持するすべてのリモート・キュー定義ごとに呼び出されます。つまり、どのメッセージが各キューから伝送されるかをルールが決定できるという意味です。

以下のルールがキューからのメッセージ伝送を許可するのは、キューのデフォルトの優先順位が 5 より大きい場合だけです。 (メッセージがキューに置かれる前に優先順位が割り当てられていない場合は、キューのデフォルトの優先順位が与えられます。)

```
public boolean transmit( MQeQueue queue )
{
  if ( queue.getDefaultPriority() > 5 )
    return (true);
  else
    return (false);
}
```

このルールを拡張して、すべてのメッセージをオフピーク時に伝送できるようにするのは実際的な方法です。そうすれば、優先順位の高いキューのメッセージだけをピーク期間に伝送することができます。次の例では、同様のアイデアをインプリメントするルールを示します。

この例では、キューに含まれるメッセージが 10 を超えた場合にのみ、メッセージの伝送を許可します。

```
public boolean transmit( MQeQueue queue )
{
    if ( queue.getNumberOfMessages() >= 10 )
```

```
    return (true);
  else
    return (false);
}
```

次の複雑な例は、伝送に要する時間に対して課金する通信ネットワーク上で、メッセージの伝送が行われていることを想定しています。また、単位時間のコストが比較的に低い*低料金時間*があることも想定しています。ルールは、低料金時間になるまで、メッセージの伝送をブロックします。低料金時間中は、キュー・マネージャーが決まった間隔で起動されます。

```
import com.ibm.mqe.*;
import java.util.*;

/**
  * Example set of  queue manager Rules which trigger the transmission
  * of any messages waiting to be sent.
  *
  * These rules only trigger the transmission of messages if the current
  * time is between the values defined in the variables cheapRatePeriodStart
  * and cheapRatePeriodEnd

  * (This example assumes that transmission will take place over a
  * communication network which charges for the time taken to transmit)
 */

public class ExampleQueueManagerRules extends MQeQueueManagerRule
                                      implements Runnable
{
  /* default interval between triggers is 10 minutes                        */
  public final int triggerInterval   = 600000;
  /* cheap rate transmission period start and end times        */
  public final int cheapRatePeriodStart = 18; /* 18:00 hrs        */
  public final int cheapRatePeriodEnd   = 9; /* 09:00 hrs        */

  /* background thread reference        */
  protected Thread    th                = null;
}
```

定数 cheapRatePeriodStart および cheapRatePeriodEnd は、この低料金時間の範囲を定義します。この例では、低料金は夜の 18:00 時から次の日の朝の 09:00 時までと定義されています。

```
/* cheap rate transmission period start and end times         */
public final int cheapRatePeriodStart = 18; /* 18:00 hrs */
public final int cheapRatePeriodEnd = 9; /* 09:00 hrs */
```

定数 triggerInterval は、キュー・マネージャーが毎回起動されるまでの間の時間を定義します (ミリ秒)。

```
public final int triggerInterval = 600000;
```

この例では、トリガー間隔は 600,000 ミリ秒と定義されています。これは、600 秒つまり 10 分と同じです。

キュー・マネージャーの起動は、 **トリガー間隔**期間の終わりに覚せいするバックグラウンド・スレッドによって処理されます。現在の時刻が低料金時間内であれば、それは **MQeQueueManager.triggerTransmission()** ルールを呼び出して、伝送を待機しているすべてのメッセージの伝送試行を開始します。

バックグラウンド・スレッドは、**queueManagerActivate()** ルールで作成され、**queueManagerClose()** ルールで停止します。キュー・マネージャーがこれらのルールを呼び出すのは、それが活動化され、クローズされたときです。

```
/**
 * Overrides MQeQueueManagerRule.queueManagerActivate()
 * Starts a timer thread
 */
 public void queueManagerActivate()
 {
   /* background thread which triggers XmitQ */
   th = new Thread( this );
   th.start();                    /* start timer thread */
 }
/**
 * Overrides MQeQueueManagerRule.queueManagerClose()
 * Stops the timer thread
 */
 public void queueManagerClose()
 {
   th.stop();                     /* stop timer thread                    */
 }
```

次は、バックグラウンド・スレッドを処理するコードの例です。

```
/**
 * Timer thread
 * Triggers queue manager every interval until thread is stopped
 */
  public void run( )
 {
  try
   {
     while ( true )
     { /* sleep for specified interval                                    */
       Thread.sleep( triggerInterval );
       /* if cheap rate period call queue manager to trigger transmission */
     if ( timeToTransmit() )
         ((MQeQueueManager)owner).triggerTransmission();
     }
   }
   catch ( Exception e )
```

```
    {
     e.printStackTrace( System.err );
    }
  }
```

変数の所有者はクラス MQeRule によって定義されます。これは、MQeQueueManagerRule の祖先です。その始動プロセスの一部として、キュー・マネージャーはキュー・マネージャー・ルールを活動化し、それ自体への参照をルール・オブジェクトに渡します。その後、この参照は変数所有者に保管されます。

スレッドは絶えずループし (**queueManagerClose()** ルールによって停止されることに注意)、トリガー間隔期間の最後になるまでスリープします。トリガー間隔の終わりに、このスレッドは **timeToTransmit()** メソッドを呼び出して、現在の時刻が低料金伝送時間内にあるかどうかを検査します。このメソッドが成功すると、キュー・マネージャーの **triggerTransmission()** ルールが呼び出されます。

timeToTransmit メソッドは、次のコードで示されます。

```
protected boolean timeToTransmit()
{
  /* get current time */
  long currentTimeLong = System.currentTimeMillis();

  Date date = new Date( currentTimeLong );
  Calendar calendar = Calendar.getInstance();
  calendar.setTime( date );

  /* get hour */
  int hour = calendar.get( Calendar.HOUR_OF_DAY );

  if ( hour >= cheapRatePeriodStart || hour < cheapRatePeriodEnd )
    return true; /* cheap rate */
  else
    return false; /* not cheap rate */
}
```

非同期リモート・キュー定義の活動化

キュー・マネージャーは起動時に、その非同期リモート・キュー定義を活動化することができます。キューを活動化するとは、キューに含まれているメッセージの伝送を試行することを意味します。この動作は、**activateQueues()** ルールによって構成することができます。

基本的なルールでは、`true` または `false` を戻すだけです。

```
public boolean activateQueues()
{
  return true; /* always transmit on activate */
}
```

他のルールと同様に、現在の時刻が低料金時間内かどうかをチェックすることができます。

```
public boolean activateQueues()
{
  if ( timeToTransmit() )
    return true;
  else
    return false;
}
```

このルールは、起動時にホーム・サーバー・キューおよびストア・アンド・フォワード (蓄積交換) キューが活動化されるかどうかも判別します。

activateQueues() が `false` を戻す場合には、メッセージがリモート・キュー定義に書き込まれるときに限り、リモート・キュー定義が活動化されます。ホーム・サーバー・キューは、キュー・マネージャーの **triggerTransmission()** ルールを呼び出すことによって、活動化することができます。

キュー・ルール

それぞれのキューには、独自のルールのセットがあります。ソリューションは、これらのルールの動作を拡張することができます。すべてのキュー・ルールは、com.ibm.mqe.MQeQueueRule の子孫でなければなりません。

キュー・ルールが呼び出されるのは、次の場合です。

- キューが活動化された。
- キューがクローズされた。
- メッセージがキューに置かれた (Put)。
- メッセージがキューから除去された (Get)。
- メッセージがキューから削除された (Delete)。
- キューがブラウズされた。
- 取り消し操作がキュー上のメッセージに対して実行された。
- メッセージ・リスナーがキューに追加された。
- メッセージ・リスナーがキューから除去された。
- メッセージの有効期限が切れた。
- キューの使用回数が変更された。
- キューの属性 (認証プログラム、暗号機能、圧縮機能) を変更しようとした。
- メッセージの索引項目が作成された。

索引項目ルール

キューはメモリーにすべてのメッセージを保持するわけではありません。メッセージはキュー・ストアに保管し、必要なときにメモリーへ復元します。キューはそのキュー・ストア内に保持しているメッセージごとに索引項目を保守します。索引項目には、ロックされているかアンロックされているかといった、メッセージの状態情報が含まれています。また、メッセージからの特定のフィールド (*索引フィールド* と呼ばれる) が、索引項目に保管されます。デフォルトの索引フィールドは、メッセージ*固有 ID*、 メッセージ *ID*、*相関 ID*、およびメッセージ*優先順位*です。これらのフィールドが保管されるのは、ほとんどのメッセージに存在するためであり、それらのフィールドをメモリーに保管することにより、より高速なメッセージ検索が可能になります。

索引項目が作成されると、**indexEntry()** ルールが呼び出されます。新規メッセージがキューに書き込まれるときには必ずこれが行われます。また、キューが活動化されるときや、キューが以前のセッションからキュー・ストアに残っているメッセージを読み取るときにも行われます。このルールにより、ソリューションは、作成時に索引項目を変更することができます。この 1 つの使用法は、共通に使用される追加フィールドを索引に追加して、メッセージ検索時間を短縮させることです。

```
/* if the message contains a customer number field - then add this field */
/* to the message's index entry.  */
/* This will enable faster message searching */
public void indexEntry( MQeFields entry,
                                MQeMsgObject msg ) throws Exception
{
  if ( msg.contains( "Cust_No" ) )
    entry.copy( msg, true, "Cust_No" );
}
```

パラメーター *entry* には、メッセージのブランク索引項目が入ります。 **indexEntry** ルールが戻った後、デフォルトの索引フィールドが、キューによって追加されます。上記の例では、メッセージに *Cust_No* というフィールドが含まれる場合、これがメッセージの索引項目に追加されます。

取得やブラウズのような、後続のメッセージング操作において、アプリケーションは *Cust_No* フィールドを、操作に提供されるフィルターの一部として使用することができます。アプリケーションが値 "75" を持つ *Cust_No* フィールド、および値 "115" を持つ *Order_No* フィールドを含むメッセージを検出しようとしているとします。この場合、キューはまず索引項目を検査し、値が "75" の *Cust_No* フィールドを含むメッセージだけをメモリーにロードして、それらが正しい値を持つ *Order_No* フィールドを含んでいるかどうかを調べることができます。 *Cust_no* フィールドが索引の一部でない場合には、すべてのメッセージがメモリーにロードされ、それがフィルターに一致するフィールドを含んでいるかどうかが調べられます。

もちろん、索引フィールドの使用は妥協案です。それらを使用してメッセージ検索時間を短縮することができます (これは、パーベイシブ・デバイスにおいて奨励されます) が、索引フィールドがメモリーに保持されます。

メッセージ有効期限切れルール

有効期限は、キューとメッセージのどちらにも設定することができます。この期限を過ぎると、そのメッセージには*期限切れ*のフラグが立てられます。その時点で、**messageExpired()** ルールが呼び出されます。メッセージに対して行われる処理はこのルールによって決まります。通常なら、これらのメッセージは削除されるか、または送達不能キューに置かれますが、このルールでは別の方法がとられます。たとえば、このルールでは、キュー上にメッセージを完全な状態で残して、 MQSeries Everyplace アプリケーションに見えるようにしておくことができます。

```
/* This rule puts a copy of any expired messages to a Dead Letter Queue */
public boolean messageExpired( MQeFields entry,
                               MQeMsgObject msg ) throws Exception
{
  /* Get the reference to the Queue Manager */
  MQeQueueManager qmgr = MQeQueueManager.getReference(
                                    ((MQeQueue)owner).getQueueManagerName() );
  /* need to set re-send flag so that put of message to new queue isn't */
  /* rejected */
  msg.putBoolean( MQe.Msg_Resend, true );
  /* if the message contains an expiry interval field - remove it */
  if ( msg.contains( MQe.Msg_ExpireTime )
    msg.delete( MQe.Msg_ExpireTime );
  /* put message onto dead letter queue */
  qmgr.putMessage( null, MQe.DeadLetter_Queue_Name, msg, null, 0 );
  /* return true & the message will be deleted from the queue */
  return (true);
}
```

前述の例では、有効期限が切れたメッセージがキュー・マネージャーの送達不能キューに送信されます。そのキューの名前は、定数 MQe.DeadLetter_Queue_Name によって定義されています。ここでの注意事項として、キュー・マネージャーは、すでに別のキューに書き込まれているメッセージの書き込みを拒否します。これにより、重複したメッセージが MQSeries Everyplace ネットワークに入ることが妨げられます。したがって、送達不能キューにメッセージを移す前に、その再送フラグを設定する必要があります。これは、メッセージに *MQe.Msg_Resend* フィールドを追加することによって行われます。また、メッセージを送達不能キューに移す前に、その有効期限フィールドを削除する必要があります。

true の値を戻すと、ルールがメッセージの有効期限が切れたと判別したことがキューに通知されます。

追加のメッセージ・リスナー・イベントのログ記録

次の例は、キュー上で発生するイベントをログに記録する方法を示しています。この例で発生するイベントは、メッセージ・リスナーの作成ですが、メッセージの書き込みやメッセージ要求のブラウズといった他のキュー・イベントの場合も、基本的なことは同じです。

例では、キューに独自のログ・ファイルがありますが、すべてのキューで使用される中央ログ・ファイルを持っているのと効果は同じです。キューは、活動化されたときにはログ・ファイルを開く必要があり、またキューがクローズされるときにはログ・ファイルをクローズする必要があります。キュー・ルール **queueActivate** および **queueClose** はこれに使用できます。どちらのルールもログ・ファイルにアクセスできるように、変数 *logFile* はクラス変数である必要があります。

```
/* This rule logs the activation of the queue */
public void queueActivate()
{
  try
  {
    logFile = new LogToDiskFile( ¥¥log.txt );
    log( MQe_Log_Information, Event_Queue_Activate, "Queue " +
         ((MQeQueue)owner).getQueueManagerName() + " + " +
         ((MQeQueue)owner).getQueueName() + " active" );
  }
  catch( Exception e )
  {
    e.printStackTrace( System.err );
  }
}
/* This rule logs the closure of the queue */
public void queueClose()
{
  try
  {
    log( MQe_Log_Information, Event_Queue_Closed, "Queue " +
         ((MQeQueue)owner).getQueueManagerName() + " + " +
         ((MQeQueue)owner).getQueueName() + " closed" );
    /* close log file */
    logFile.close();
  }
  catch ( Exception e )
  {
    e.printStackTrace( System.err );
  }
}
```

addListener ルールは、次のコードで示されます。これは **MQe.log** メソッドを使用して、Event_Queue_AddMsgListener イベントを追加します。

```
/* This rule logs the addition of a message listener */
public void addListener( MQeMessageListenerInterface listener,
                         MQeFields filter ) throws Exception
{
  log( MQe_Log_Information, Event_Queue_AddMsgListener,
       "Added listener on queue " +
       ((MQeQueue)owner).getQueueManagerName() + "+" +
       ((MQeQueue)owner).getQueueName() );
}
```

第6章 メッセージング・リソースの管理

キュー・マネージャーやキューのような MQSeries Everyplace リソースの管理は、専用の MQSeries Everyplace メッセージを使用して行われます。メッセージを使用するなら、管理はローカルでもリモートでも行うことができます。

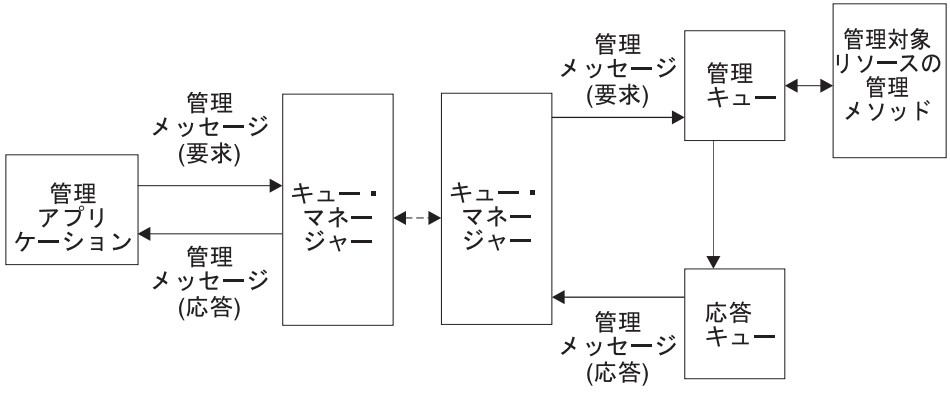

図 13. MQSeries Everyplace 管理

キュー・マネージャーやそのリソースを管理をするためには、キュー・マネージャーを開始し、その上に管理キューを構成する必要があります。管理キューには、キューが受け取った管理メッセージを順番に処理するという役割があります。 1 度に処理できる要求は 1 つだけです。キューは、 MQeQueueManagerConfigure クラスの **defineDefaultAdminQueue()** メソッドを使用して作成することができます。キューの名前は `AdminQ` であり、アプリケーションはこれを定数 `MQe.Admin_Queue_Name` を使用して参照することができます。

一般的な管理アプリケーションでは、まず MQeAdminMsg のサブクラスをインスタンス化し、これを必要な管理要求と共に構成して宛先キュー・マネージャーの `AdminQ` に渡します。アプリケーションがアクションの結果を知る必要がある場合は、応答を要求できます。要求が処理されると、要求メッセージで指定した応答先キューとキュー・マネージャーに、要求の結果を示すメッセージが返されます。

応答は任意のキュー・マネージャーまたはキューに送ることができますが、デフォルトの応答先キューを構成して、管理応答メッセージ専用のキューにすることもできます。このデフォルト・キューは、 MQeQueueManagerConfigure クラスの

defineDefaultAdminReplyQueue() メソッドを使用して作成されます。キューの名前は AdminReplyQ であり、アプリケーションはこれを定数 MQe.Admin_Reply_Queue_Name を使用して参照することができます。

管理キューには、個々のリソースの管理を実行する方法についての情報はありません。この情報は、各リソースおよびそれに対応する管理メッセージにカプセル化されます。MQSeries Everyplace リソースの管理においては、次のメッセージが使用されます。

表 3. 管理メッセージ

メッセージ名	目的
MQeAdminMsg	すべての管理メッセージの基底クラスとして使用される要約クラス
MQeAdminQueueAdminMsg	管理キューの管理をサポートする
MQeConnectionAdminMsg	キュー・マネージャー間の接続の管理をサポートする
MQeHomeServerQueueAdminMsg	ホーム・サーバー・キューの管理をサポートする
MQeQueueAdminMsg	ローカル・キューの管理をサポートする
MQeQueueMangerAdminMsg	キュー・マネージャーの管理をサポートする
MQeRemoteQueueAdminMsg	リモート・キューの管理をサポートする
MQeStoreAndForwardQueueAdminMsg	ストア・アンド・フォワード (蓄積交換) キューの管理をサポートする
MQeMQBridgeQueueAdminMsg	MQSeries システムに接続するキューの管理をサポートする

これらの基本管理メッセージは、 com.ibm.mqe.administration パッケージの中に提供されています。その他のタイプやリソースは、 MQeAdminMsg またはいずれかの既存の管理メッセージをサブクラス化することによって管理できます。たとえば、MQSeries-ブリッジ の管理のための追加の管理メッセージは、 com.ibm.mqe.mqbridge パッケージの中にあります。

基本となる管理要求メッセージ

MQSeries Everyplace リソースの管理要求には、すべて同じ基本の形式があります。121ページの図14 は、すべての管理要求メッセージに使用される基本的な構造を示しています。

要求は次のものから成っています。

1. すべての管理要求に共通する、管理固有のフィールド
2. 管理されるリソースに固有な、エラーの詳細を示す管理のフィールド
3. 管理メッセージの処理を容易にする、オプションのフィールド

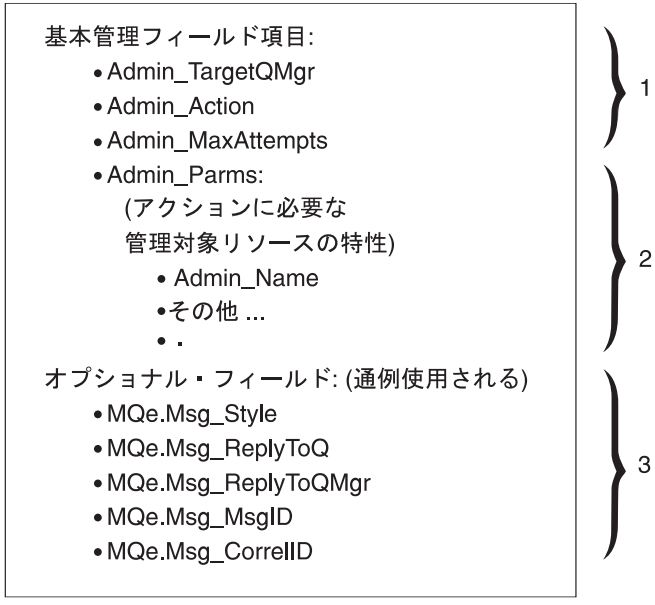

図 14. 管理要求メッセージ

基本管理フィールド

すべての管理要求に共通する、管理固有のフィールド

Admin_Target_QMgr

このフィールドでは、要求されるアクションを実行するキュー・マネージャー名 (宛先キュー・マネージャー) を設定します。宛先キュー・マネージャーは、ローカル・キュー・マネージャーでも、リモート・キュー・マネージャーのでも構いません。 Java 仮想マシンでは、1 度に 1 つのキュー・マネージャーしかアクティブにすることができないため、メッセージの置かれるキュー・マネージャーが宛先キュー・マネージャーとなります。

Admin_Action

このフィールドには、実行する管理アクションが入ります。各管理対象リソースでは、一連の管理アクションを実行することができますが、管理メッセージで要求できるのは、実行される 1 つのアクションだけです。次のような共通のアクションが定義されています。

表 4. 管理アクション

管理アクション	目的
Action_Create	管理対象リソースの新しいインスタンスを作成する

表 4. 管理アクション (続き)

管理アクション	目的
Action_Delete	既存の管理対象リソースを削除する
Action_Inquire	管理対象リソースの 1 つまたは複数の特性を照会する
Action_InquireAll	管理対象リソースのすべての特性を照会する
Action_Update	管理対象リソースの 1 つまたは複数の特性を更新する

これらのアクションは、必ずしもすべてのリソースでインプリメントする必要はありません。たとえば、管理メッセージを使用してキュー・マネージャーを作成することはできません。特定の管理メッセージでは、基本のセットを拡張して、リソースに固有の付加的なアクションを実行できるようにすることもできます。

それぞれの共通アクションは、次のメソッドで Admin_Action フィールドに設定することができます。

表 5. 管理アクション・フィールドの設定

管理アクション	設定メソッド
Action_Create	create(MQeFields parms)
Action_Delete	delete(MQeFields parms)
Action_Inquire	inquire(MQeFields parms)
Action_InquireAll	inquireAll(MQeFields parms)
Action_Update	update(MQeFields parms)

Admin_MaxAttempts

このフィールドでは、最初のアクションが失敗した場合に、そのアクションを再試行できる回数を指定します。再試行は、キュー・マネージャーの次の始動時、もしくは、管理キューに設定されている次のインターバルに実行されます。

その他のフィールド

ほとんどの失敗では、応答メッセージの中で詳細な情報を確認することができます。失敗の情報の読み取りと処理は、要求を出したアプリケーションで行われます。応答データの使用に関する詳細については、126ページの『基本となる管理応答メッセージ』を参照してください。

次の一連のメソッドを使って、いくつかの要求のフィールドを設定することができます。

表 6. 管理要求フィールドの設定

管理アクション	フィールド・タイプ	set および get メソッド
Admin_Parms	MQeFields	MQeFields getInputFields()

表 6. 管理要求フィールドの設定 (続き)

管理アクション	フィールド・タイプ	set および get メソッド
Admin_Action	int	setAction(int action)
Admin_TargetQMgr	ascii	setTargetQMgr(String qmgr)
Admin_MaxAttempts	int	setMaxAttempts(int attempts)

管理対象ノードに固有のフィールド

Admin_Parms

このフィールドには、アクションで必要とされるリソース特性が入ります。

各リソースには、固有な一連の特性があります。それぞれの特性には名前とタイプと値があります。そして、各特性の名前は管理メッセージの中の定数によって定義されます。リソースの名前は、すべての管理対象リソースに共通する特性です。リソースの名前は、 *Admin_Name* にあり、そのタイプは ascii です。

一連のリソース特性をすべて判別するには、管理メッセージのインスタンスに対して **characteristics()** メソッドを使用します。このメソッドを使用すると、各特性ごとに 1 つのフィールドを含む MQeFields オブジェクトが返されます。各特性の名前、タイプ、およびデフォルト値を確認するには、MQeFields メソッドを使用して一連の特性を列挙することができます。

アクションに渡すことのできる一連の特性は、要求されるアクションによって異なりますが、リソースの名前 *Admin_Name* だけは、どんな場合でも必ず渡されなければなりません。なお、**Action_InquireAll** が要求される場合は、このパラメーターが唯一の必須パラメーターとなります。

管理メッセージで管理するリソースの名前を設定する場合は、次のようなコードが使用されます。

```
SetResourceName( MQeAdminMsg msg, String name )
{
  MQeFields parms;
  if ( msg.contains( Admin_Parms ) )
    parms = msg.getFields( Admin_Parms );

  else
    parms = new MQeFields();

  parms.putAscii( Admin_Name, name );
  msg.putFields( Admin_Parms, name );
}
```

あるいは別の方法として、メッセージから *Admin_Parms* フィールドを返す **getInputFields()** メソッドか、メッセージに *Admin_Name* フィールドを設定する **setName()** を使用するなら、コードを簡単にすることができます。これは、次のコードで示されます。

```
SetResourceName( MQeAdminMsg msg, String name )
{
  msg.SetName( name );
}
```

他の便利なフィールド

デフォルトでは、管理要求が処理されたときに応答は行われません。応答が必要な場合は、応答のメッセージを求めるように要求のメッセージをセットアップする必要があります。 MQe クラスでは、応答の要求に使用される次のようなフィールドが定義されています。

Msg_Style
: 次の 3 つの値のいずれかをとれる int タイプのフィールド

  Msg_Style_Datagram
  : 応答を要求しないコマンド

  Msg_Style_Request
  : 応答を求める要求

  Msg_Style_Reply
  : 要求に対する応答

Msg_Style を Msg_Style_Request (応答を要求する) に設定した場合は、要求メッセージの中で応答先の位置を設定する必要があります。応答先の設定には、次の 2 つのフィールドを使用します。

Msg_ReplyToQ
: 応答用のキューの名前を保持する ascii フィールド

Msg_ReplyToQMgr
: 応答用のキュー・マネージャーの名前を保持する ascii フィールド

応答先のキュー・マネージャーと要求を処理するキュー・マネージャーが異なる場合、要求を処理するキュー・マネージャーには応答先のキュー・マネージャーへの接続を定義する必要があります。

管理要求メッセージをその応答メッセージと相互に関連付ける場合は、要求を一意的に識別し、かつ応答メッセージにコピーすることのできるフィールドをその要求メッセージに含める必要があります。 MQSeries Everyplace には、この目的で使用することのできる、 2 つのフィールドがあります。

Msg_MsgID
: メッセージ ID を含むバイト配列

Msg_CorrelID
: メッセージの相関 ID を含むバイト配列

要求メッセージに含めるフィールド自体は他のどんなフィールドでも構いませんが、この 2 つのフィールドが持つ大きな利点として、キュー・マネージャーは、これらのフィールドを使用した場合に最も効率的にキューやメッセージを検索することができます。以下のコード・フラグでは、要求メッセージを作成する方法の例を示しています。

```
public class LocalQueueAdmin extends MQe
{
  public String   targetQMgr = "ExampleQM";  // target queue manager
public MQeFields primeAdminMsg(MQeAdminMsg msg) throws Exception
{
  /*
   * Set the target queue manager that will process this message
   */
  msg.setTargetQMgr( targetQMgr );

  /*
   * Ask for a reply message to be sent to the queue
   * manager that processes the admin request
   */
  msg.putInt  (MQe.Msg_Style,       MQe.Msg_Style_Request);
  msg.putAscii(MQe.Msg_ReplyToQ,    MQe.Admin_Reply_Queue_Name);
  msg.putAscii(MQe.Msg_ReplyToQMgr, targetQMgr);

  /*
   * Setup the correl id so we can match the reply to the request.
   * - Use a value that is unique to the this queue manager.
   */
  byte[] correlID = Long.toHexString( (MQe.uniqueValue()).getBytes() );
  msg.putArrayOfByte( MQe.Msg_CorrelID, correlID );

  /*
   * Ensure matching response message is retrieved
   * - set up a fields object that can be used as a match parameter
   *   when searching and retrieving messages.
   */
  MQeFields msgTest = new MQeFields();
  msgTest.putArrayOfByte( MQe.Msg_CorrelID, correlID );

  /*
   * Return the unique filter for this message
   */
  return msgTest;
}
```

管理要求のメッセージを作成したら、通常の MQSeries Everyplace メッセージ処理 API を使用してこれを宛先キュー・マネージャーに送ります。メッセージを同期で送達できるか非同期で送達できるかは、宛先の管理キューをどのように定義するかによって異なります。

応答の待機または応答の通知のために、 API を処理する 標準 MQSeries Everyplace メッセージも使用されます。要求の送信と応答メッセージの受信との間には時間のずれが生じます。この時間のずれは、要求がローカルに処理される場合には短く、要求も応答

メッセージも共に非同期で送達される場合には長くなります。次のコード・フラグを使用して、要求メッセージを送信し、その応答を待つことができます。

```
public class LocalQueueAdmin extends MQe
{
  public String   targetQMgr = "ExampleQM";  // target queue manager
  public int      waitFor    =   10000;      // millsecs to wait for reply

/*
 * Send a completed admin message.
 *   Uses the simple putMessage method which is not assured if the
 *   the queue is defined for synchronous operation.
 */
public void sendRequest( MQeAdminMsg msg ) throws Exception
{
  myQM.putMessage( targetQMgr,
                   MQe.Admin_Queue_Name,
                   msg,
                   null,
                   0 );
}
/*
 * Wait a while for a reply message. This method will wait for
 * a limited time on either a local or a remote reply to queue.
 * Parameters:
 *   msgTest: a filter for the reply message to wait for
 * Returns:
 *   respMsg: a reply message matching the msgTest filter.
 */
public MQeAdminMsg waitForReply( MQeFields msgTest ) throws Exception
{
  MQeAdminMsg respMsg = null;
  respMsg = (MQeAdminMsg)myQM.waitForMessage(targetQMgr,
                                             MQe.Admin_Reply_Queue_Name,
                                             msgTest,
                                             null,
                                             0,
                                             waitFor);
  return respMsg;
}
```

基本となる管理応答メッセージ

管理要求が処理されると、応答 (要求された場合) が応答先のキュー・マネージャー・キューに送られます。応答メッセージは、要求メッセージと同じ基本の形式を持っていて、要求メッセージよりもいくつか多くのフィールドを含んでいます。

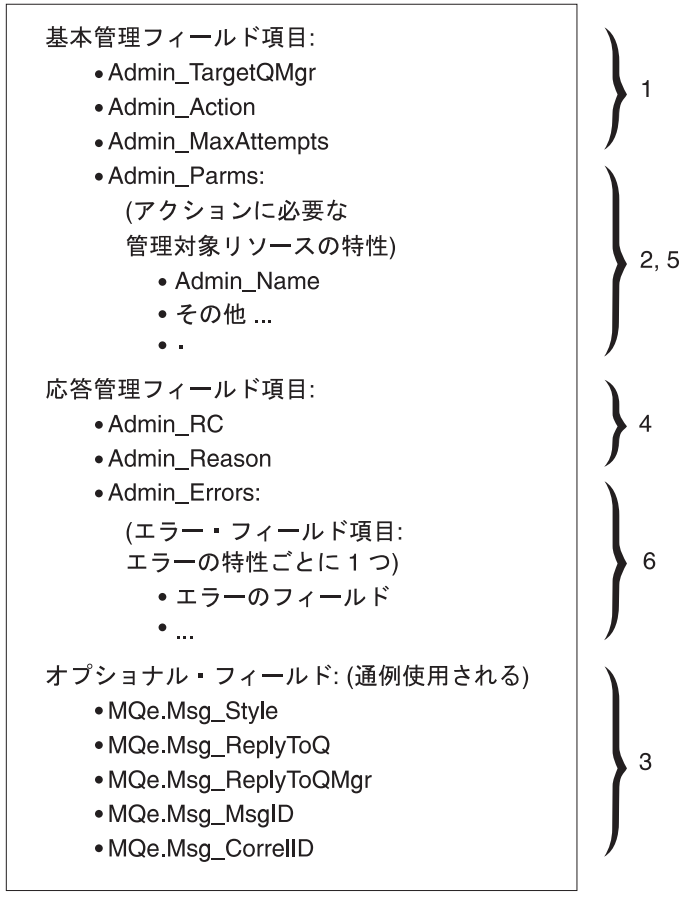

図 15. 管理応答メッセージ

応答は次のものから成っています。

1. 基本管理フィールド (要求メッセージからコピーされる)
2. 管理されるリソースに固有な、エラーの詳細を示す管理のフィールド
3. 管理メッセージの処理を容易にする、オプションのフィールド (要求メッセージからコピーされる)
4. 要求の結果を詳細に記述する管理のフィールド
5. 管理されるリソースに固有な、要求の結果を詳細に記述する管理のフィールド
6. 管理されるリソースに固有な、エラーの詳細を示す管理のフィールド

最初の 3 つの項目については 120ページの『基本となる管理要求メッセージ』で説明しています。応答の固有フィールドについては、次のセクションで説明します。

要求の結果フィールド

Admin_RC field

このバイト・フィールドには、要求の全体的な結果が含まれています。これは int タイプのフィールドで、次の 3 つのうちのいずれかの値をとります。

MQeAdminMsg.RC_Success
アクションは正常に完了。

MQeAdminMsg.RC_Failed
要求は完全に失敗。

MQeAdminMsg.RC_Mixed
要求は部分的に成功。 4 つのキューの属性を更新する要求が出され、3 つの属性で更新が正常に完了したがその 1 つで要求が失敗した場合は、戻りコード Mixed が返されます。

Admin_Reason
Mixed および Failed が返された場合に、その失敗についての全体的な理由が記される unicode フィールド。

Admin_Parms
管理対象ノードの各特性ごとのフィールドを含む MQeFields オブジェクト。

Admin_Errors
失敗した各更新ごとのフィールドを含む MQeFields オブジェクト。
Admin_Errors フィールドに含まれる各項目は、 ascii タイプか asciiArray タイプです。

次のメソッドを使用して、いくつかの応答のフィールドを確認することができます。

表 7. 管理応答フィールドの確認

管理フィールド	フィールド・タイプ	確認メソッド
Admin_RC	int	int getAction()
Admin_Reason	unicode	String getReason()
Admin_Parms	MQeFields	MQeFields getOutputFields()
Admin_Errors	MQeFields	MQeFields getErrorFields()

実行されるアクションによっては、関係するフィールドだけが、戻りコードや理由として返されます。これは **delete** アクションの場合に当てはまります。一方 **inquire** のような他のアクションの場合、応答メッセージには、もっと詳細な情報が要されます。たとえば、フィールド *Queue_Description* および *Queue_FileDesc* に対して **inquire** 要求が出された場合、結果の MQeFields オブジェクトには、これら 2 つのフィールドに対する実際のキューの値が含まれます。

次の表は、要求メッセージの *Admin_Parms* フィールドと、キューのいくつかのパラメーターでの inquire に対する応答メッセージのフィールドを示したものです。

表 8. キュー・パラメーターの照会

Admin_Parms フィールド名	**要求メッセージ**		**応答メッセージ**	
	タイプ	**値**	**タイプ**	**値**
Admin_Name	ascii	″TestQ″	ascii	″TestQ″
Queue_QMgrName	ascii	″ExampleQM″	ascii	″ExampleQM″
Queue_Description	Unicode	ヌル	Unicode	″A test queue″
Queue_FileDesc	ascii	ヌル	ascii	″c:¥queues¥″

応答で付加的なデータが予期されないアクションでは、応答の *Admin_Parms* フィールドと要求メッセージの同じフィールドが一致します。これは、**create** アクションや **update** アクションの場合に当てはまります。

create や **update** のようないくつかのアクションでは、管理対象リソースの複数の特性を設定または更新するよう要求する場合があります。その場合、戻りコード RC_Mixed が受信される可能性があります。それぞれの更新がなぜ失敗したかを示す追加の詳細は、 *Admin_Errors* フィールドで確認することができます。次の表は、キューの更新の要求に対する *Admin_Parms* フィールドと結果の *Admin_Errors* フィールドの例を示したものです。

表 9. キューの更新のための要求および応答メッセージ

フィールド名	**要求メッセージ**		**応答メッセージ**	
	タイプ	**値**	**タイプ**	**値**
Admin_Parms フィールド				
Admin_Name	ascii	″TestQ″	ascii	″TestQ″
Queue_QMgrName	ascii	″ExampleQM″	ascii	″ExampleQM″
Queue_Description	Unicode	ヌル	Unicode	″ExampleQM″ ″A new description″
Queue_FileDesc	ascii	ヌル	Unicode	″D:¥queues″
Admin_Errors フィールド				
Queue_FileDesc	n/a	n/a	ascii	″Code=4;com.ibm.mqe.MQeException: wrong field type″

更新や設定が正常に行われたフィールドについては、*Admin_Errors* フィールドに項目は存在しません。

各エラーの詳細な記述は、ascii ストリングで戻されます。エラー・ストリングの値は、設定か更新が試行された際に発生した例外です。例外が MQeException であった場合、

実際の例外コードは、例外の *toString* 表示と共に返されます。したがって、MQeException の場合の値の形式は次のようになります。

```
"Code=nnnn;toString representation of the exception"
```

次のコード・フラグは、管理要求の結果を調べ、すべてのエラーを System.out に送る方法を示しています。

```
/**
 * Check to see if a good reply was received.
 * If not detail the error(s) that occurred
 * @return boolean  true if good
 * @param replyMsg reply message to check
 * Throws an Exception if the request failed.
 */
public boolean checkReply( MQeAdminMsg replyMsg ) throws Exception
{
  // Was a reply received ?
  if (replyMsg == null)
  {
    System.out.println("..No response received to the request");
    throw new Exception("No response message received");
  }
  // If the reply was not successful output details for failure
  if ( replyMsg.getRC() != MQeAdminMsg.RC_Success)
  {
    System.out.println("..Action Failed: "+replyMsg.getReason());

    // If mixed then detail each error that occurred
    if ( replyMsg.getRC() == MQeAdminMsg.RC_Mixed)
    {
      MQeFields errors = replyMsg.getErrorFields();
      Enumeration en = errors.fields();
      // process each error
      while( en.hasMoreElements() )
      {
        String value[];
        String name = (String)en.nextElement();
        // Field in error may be an array
        if ( errors.dataType( name ) == MQeField.TypeArrayElements )
          value = errors.getAsciiArray( name );
        else
          value = new String[] { errors.getAscii( name ) };
        for (int j=0; j<value.length; j++)
          System.out.println("Field in error: "+name+"  "+value[j]);
      }
    }
    // Request failed so throw exception
    throw new MQeException(replyMsg.getReason());
  }
  return true;    // All is OK
}
```

管理対象リソースの管理

前のセクションで扱ったように、 MQSeries Everyplace には、管理メッセージで管理できる一連のリソースがあります。これらのリソースは、*管理対象リソース* として知られています。続くセクションでは、これらのリソースのいくつかを管理する方法について扱います。各リソースのアプリケーション・プログラミング・インターフェースに関する詳細は、「*MQSeries Everyplace for Multiplatforms プログラミング・リファレンス*」を参照してください。

キュー・マネージャー

ほとんどの管理対象リソースのライフ・サイクルの管理は、管理メッセージによって制御することができます。これはつまり、管理対象リソースは、管理メッセージによって存在するようになり、管理され、削除されることを意味します。これはキュー・マネージャーには該当しません。キュー・マネージャーを管理するためには、キュー・マネージャーがすでに作成され、開始されている必要があります。キュー・マネージャーの作成と開始については、 45ページの『キュー・マネージャーの作成および削除』を参照してください。

キュー・マネージャーそのものにはほとんど特性がありませんが、キュー・マネージャーは他の MQSeries Everyplace リソースを制御します。キュー・マネージャー上で照会を実行すると、他のキュー・マネージャーへの接続のリストや、そのキュー・マネージャーが作業できるキューのリストを確認することができます。それぞれのリスト項目には、接続またはキューの名前が示されます。リソース名が分かれば、適切なメッセージを使用してそのリソースを管理することができます。たとえば、接続の管理を行うには、 MQeConnectionAdminMessage メッセージを使用します。

接続

接続では、特定のキュー・マネージャーが別のキュー・マネージャーにどのように接続するかを定義します。接続を定義すると、キュー・マネージャーからリモート・キュー・マネージャーのキューにメッセージを書き込むことができるようになります。次の図は、別のキュー・マネージャーにあるキューと通信する上で、1 つのキュー・マネージャーのリモート・キューに必要なコンポーネントを示しています。

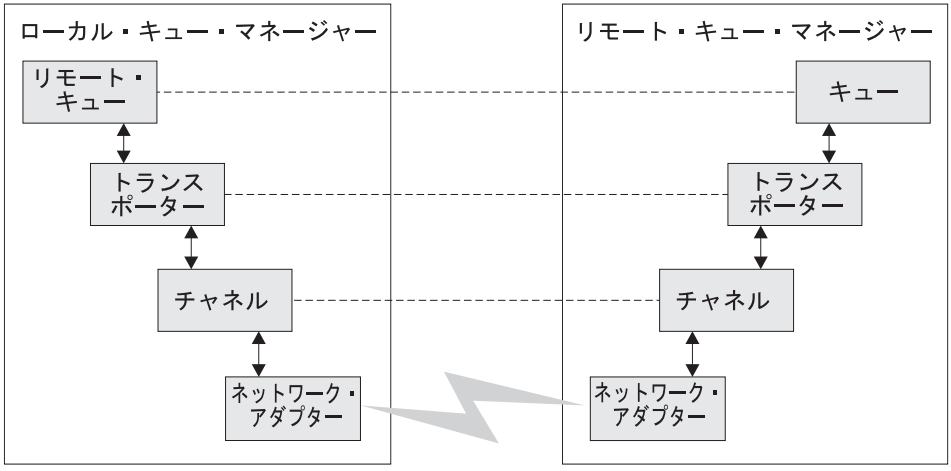

図 16. キュー・マネージャー接続

通信は、さまざまなレベルで行われます。

トランスポーター:
2 つのキューの間の論理接続

チャネル:
2 つのシステムの間の論理接続

アダプター:
プロトコル固有の接続

チャネルとアダプターは、接続の定義の過程で指定します。トランスポーターは、リモート・キューを定義する過程で指定します。次のサンプル・コードは、接続を確立できる状態の MQeConnectionAdminMsg をインスタンス化および準備するメソッドを示しています。

```
/**
 * Setup an admin msg to create a connection definition
 */
public MQeConnectionAdminMsg addConnection( remoteQMgr
  adapter,
                parms,
                options,
                channel,
                desc ) throws Exception
{
  String remoteQMgr  = "ServerQM";
  /*
   * Create an empty queue manager admin message and parameters field
   */
  MQeConnectionAdminMsg msg = new MQeConnectionAdminMsg();

  /*
```

```
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of queue manager to add routes to
   */
  msg.setName( remoteQMgr );

  /*
   * Set the admin action to create a new queue
   * The connection is setup to use a default channel. This is an alias
   * which must have be setup on the queue manager for the connection to
   * work.
   */
  msg.create( adapter,
              parms,
              options,
              channel,
              desc );

  return msg;
}
```

MQSeries Everyplace では、チャネルおよびアダプターのタイプを選択することができます。選択の内容に従って、キュー・マネージャーは次の方法で接続することができます。

- クライアントからサーバーへ
- 対等通信

クライアントからサーバーへ

クライアント / サーバー構成では、1 つのキュー・マネージャーがクライアントとして機能し、別のキュー・マネージャーがサーバー環境として稼働します。サーバーでは、複数の着信接続 (チャネル) を同時に受けることができます。このため、サーバーには複数の着信要求を処理できるコンポーネントが必要です。サーバーの環境でキュー・マネージャーを稼働させる方法については、 64ページの『サーバー・キュー・マネージャー』を参照してください。

134ページの図17 は、クライアント / サーバー構成の一般的な接続コンポーネントを示したものです。

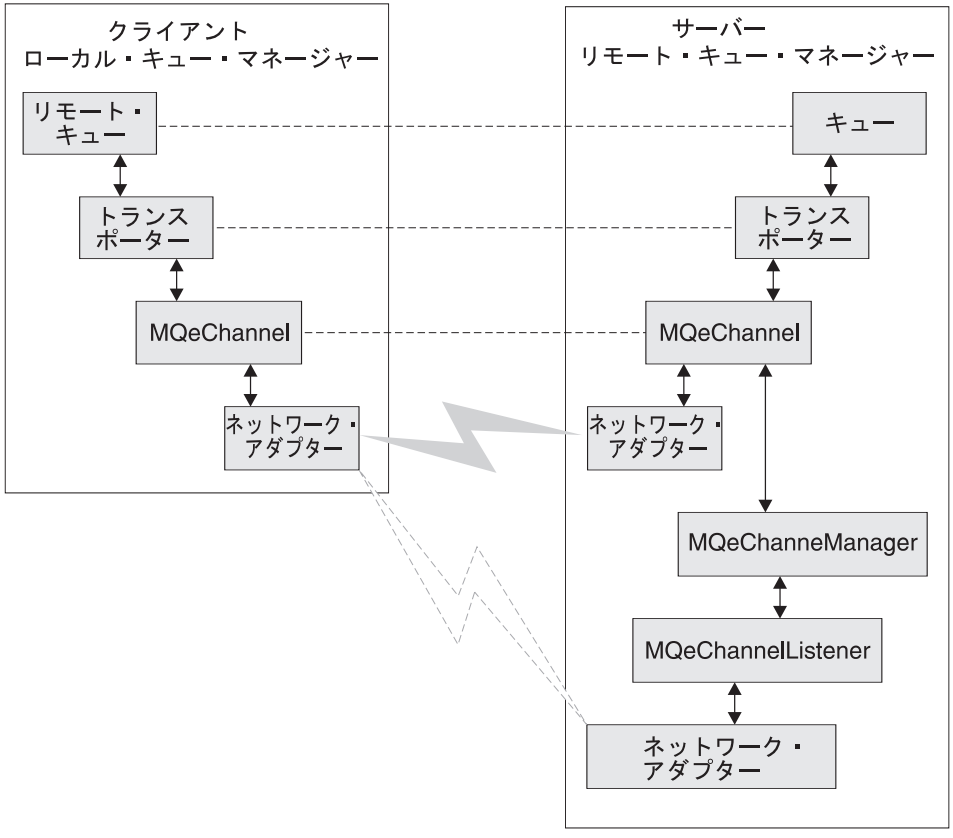

図 17. クライアント / サーバー接続

クライアント部分の接続を構成するには、MQeConnectionAdminMsg を使用します。チャネル・タイプは `com.ibm.mqe.MQeChannel` です。通常、`DefaultChannel` の別名は MQeChannel に構成されます。次のコード・フラグは、 http プロトコルを使用してサーバーと通信するクライアントで接続を構成する方法を示しています。

```
/**
 * Create a connection admin message that creates a connection
 * definition to a remote queue manager using the HTTP protocol. Then
 * send the message to the client queue manager.
 */
public addClientConnection( MQeQueueManager myQM,
  String targetQMgr ) throws Exception
{
  String remoteQMgr  = "ServerQM";
  String adapter     = "Network:127.0.0.1:80";
// This assumes that an alias called Network has been setup for
// network adapter com.ibm.mqe.adapters.MQeTcpipHttpAdapter
  String parameters  = null;
  String options     = null;
```

```
  String channel     = "DefaultChannel";
  String description = "client connection to ServerQM";

  /*
   * Setup the admin msg
   */
  MQeConnectionAdminMsg msg = addConnection( remoteQMgr,
                                             adapter,
                                             parameters,
                                             options,
                                             channel,
                                             desc );
  /*
   *  Put the admin message to the admin queue (not using assured flows)
   */
  myQM.putMessage(targetQMgr,
  MQe.Admin_Queue_Name,
  msg,
  null,
  0 );
}
```

対等通信

対等通信構成において、キュー・マネージャーは、他の多くの対等機能に対して同時に発信を行うことができるが、どんな場合にも、1 つの対等機能からしか着信を受けることができない対等機能として稼働します。対等機能の中から 1 つがマスターまたは起動側として構成され、それ以外はスレーブまたは受信側として構成されます。

マスターは、クライアント接続の定義とほとんど同じ方法で構成されます。唯一異なる点は、使用されるチャネルのタイプです。チャネル・タイプは、`com.ibm.mqe.adapters.MQePeerChannel` (または別名) に設定されなければなりません。

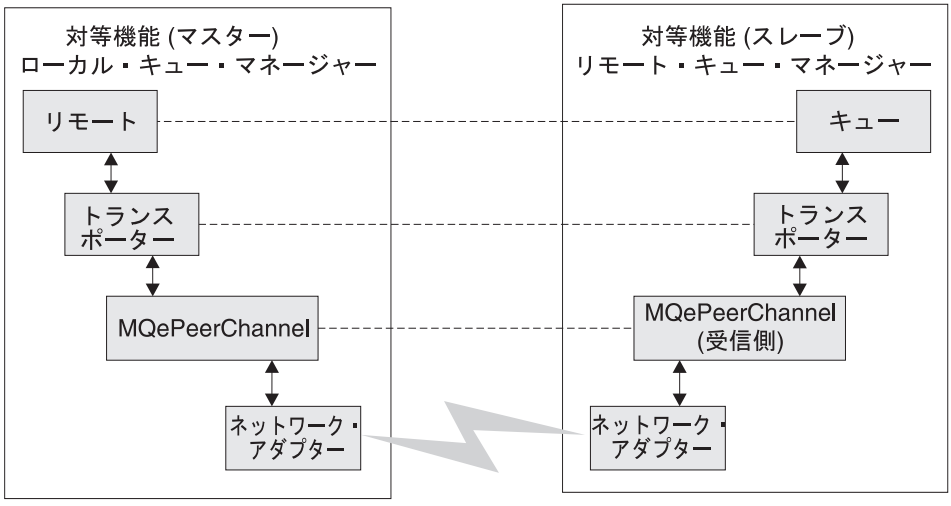

図 18. 対等通信接続

スレーブまたは受信側も同様の方法で構成されますが、以下の点が異なっています。

- 接続定義の名前が、接続を定義するキュー・マネージャーの名前と一致していなければならない
- チャネル・タイプが、`com.ibm.mqe.adapters.MQePeerChannel` でなければならない
- アダプターをリスナーとして構成しなければならない

次のコード・フラグでは、HTTP プロトコルを使用してポート 8082 で listen し、`PeerQM1` というキュー・マネージャーを対等通信の受信側として構成します。

```
/**
 * Create a connection admin message which will create a connection
 * definition to a remote queue manager using the HTTP protocol. Then
 * send the message to the client queue manager.
 */
public addClientConnection( MQeQueueManager myQM,
  String targetQMgr ) throws Exception
{
  String remoteQMgr  = "PeerQM1";
// To be a receiver the connection definition called "PeerQM1" must
// be configured on queue manager "PeerQM1"
  String adapter     = "Network::8082";
// This assumes that an alias called Network has been setup for
// network adapter com.ibm.mqe.adapters.MQeTcpipHttpAdapter
  String parameters  = null;
  String options     = null;
  String channel     = "com.ibm.mqe.adapters.MQePeerChannel";
  String description = "peer receiver on PeerQM";

  /*
   * Setup the admin msg
```

```
   */
  MQeConnectionAdminMsg msg = addConnection( remoteQMgr,
  adapter,
                parameters,
                options,
                channel,
                desc );
  /*
   *  Put the admin message to the admin queue (not using assured flows)
   */
  myQM.putMessage(targetQMgr,
  MQe.Admin_Queue_Name,
  msg,
  null,
  0 );
}
```

次の表は、PeerQM1 の受信側と、これに接続しようとする他のすべての対等キュー・マネージャーに使用される、接続定義パラメーターを示したものです。

表 10. 対等通信接続定義

	マスター (起動側)	**スレーブ (受信側)**
キュー・マネージャー	任意	″PeerQM1″
接続名	″PeerQM1″	″PeerQM1″
チャネル	com.ibm.mqe.MQePeerChannel	com.ibm.mqe.MQePeerChannel
アダプター	Network:192.168.0.10:8082	Network::8082

アダプター

MQSeries Everyplace で提供されているアダプターについての詳細は、 289ページの『第10章 MQSeries Everyplace アダプター』および MQSeries Everyplace for Multiplatforms プログラミング・リファレンス の第 9 章を参照してください。

経路指定接続

接続は、キュー・マネージャーが中間キュー・マネージャーを経由してメッセージを送れるようにセットアップすることができます。これには次の 2 つの接続が必要です。

1. 中間キュー・マネージャーへの接続
2. 宛先キュー・マネージャーへの接続

最初の接続を行うには、このセクションの初めの方で説明されている、クライアント接続や対等通信接続と同じメソッドを使用することができます。2 つ目の接続を行うには、ネットワーク・アダプター名に中間キュー・マネージャー名を指定します。このような構成において、アプリケーションは宛先キュー・マネージャーにメッセージを書き込みますが、その際、メッセージは 1 つ以上の中間キュー・マネージャーを介して宛先に送られます。

別名

接続には複数の名前や別名を割り当てることができます (59ページの『別名』を参照)。アプリケーションで MQeQueueManager クラスのメソッドを呼び出し、キュー・マネージャー名の指定が要求された際には、別名を使用することもできます。

別名は、ローカル・キュー・マネージャーでも、リモート・キュー・マネージャーでも使用することができます。ローカル・キュー・マネージャーに別名を付けるためには、まずローカル・キュー・マネージャーと同じ名前の接続定義を確立する必要があります。これは、すべてのパラメーターをヌルに設定できる論理接続です。

別名を追加および除去するには、 MQeConnectionAdminMsg クラスの **Action_AddAlias** アクションおよび **Action_RemoveAlias** アクションを使用します。1 つのメッセージで複数の別名を追加または削除することができます。 ascii 配列フィールドの *Con_Aliases* を設定することによって、操作したい別名を直接メッセージの中に書き込みます。 **addAlias()** メソッドや **removeAlias()** メソッドを使用することもできます。各メソッドはそれぞれ 1 つの別名しか扱えませんが、メソッドを繰り返し呼び出して 1 つのメッセージに複数の別名を追加できます。次のコードの抜粋は、メッセージに接続の別名を追加する方法を示すものです。

```
/**
 * Setup an admin msg to add aliases to a queue manager (connection)
 */
public MQeConnectionAdminMsg addAliases(  String queueManagerName
                                          String aliases[] ) throws Exception
{
  /*
   * Create an empty connection admin message
   */
  MQeConnectionAdminMsg msg = new MQeConnectionAdminMsg();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of the connection to add aliases to
   */
  msg.setName( queueManagerName );

  /*
   * Use the addAlias method to add aliases to the message.
   */
  for ( int i=0; i<aliases.length; i++ )
  {
    msg.addAlias( aliases[i] );
  }

  return msg;
}
```

キュー

MQSeries Everyplace で提供されているキュー・タイプについては、 4ページの『MQSeries Everyplace キュー』で簡潔に説明しています。その中で最も単純なのがローカル・キューで、このキューは、MQeQueue クラスにインプリメントされ、MQeQueueAdminMsg のクラスによって管理されます。他のすべてのタイプのキューは、 MQeQueue を継承します。各タイプのキューには、それに対応して、MQeQueueAdminMsg を継承する管理メッセージが存在します。次のセクションでは、さまざまなタイプのキューの管理について説明します。

ローカル・キュー

ローカル・キューとそれに属するキューでは、 MQSeries Everyplace で提供されている管理アクションを使用して作成、更新、削除、および照会を行うことができます。基本的な管理のメカニズムは、MQeAdminMsg から継承されます。

キューの名前は、宛先キュー・マネージャーの名前 (ローカル・キューの場合は、そのキューが属しているキュー・マネージャーの名前) からとられ、そのキュー・マネージャーのキューに固有な名前が用いられます。キューは、管理メッセージ内の 2 つの ascii フィールド、すなわち *Admin_Name* フィールドと *Queue_QMgrName* フィールドによって一意的に識別されます。これら管理メッセージ内の 2 つのフィールドは、メソッド **setName(queueManagerName, queueName)** を使用して設定することができます。

次の図は、ローカル・キューが含まれているキュー・マネージャーの例を示しています。キュー・マネージャー qm1 には、invQ というキューがあります。キューのキュー・マネージャー名特性が qm1 であり、キュー・マネージャーの名前と一致しています。

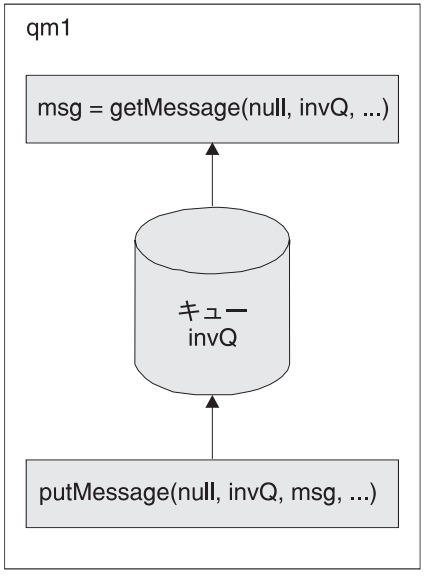

図 19. ローカル・キュー

メッセージ・ストア: ローカル・キューでメッセージを保管するには、メッセージ・ストアが必要です。各キューは、使用するストアのタイプとそのストアの場所を指定することができます。使用するメッセージ・ストアのタイプは、キューの特性 *Queue_FileDesc* とそのパラメーターによって指定することができます。このフィールドは ascii タイプのフィールドで、値は、次の形式のファイル記述子でなければなりません。

adapter class:adapter parameters
または
adapter alias:adapter parameters

たとえば、次のように入力します。

```
MsgLog:d:¥QueueManager¥ServerQM12¥Queues
```

MQSeries Everyplace V1.2 では、2 つのアダプター、すなわちディスクへのメッセージの書き込みを行うためのアダプターと、それらメッセージをメモリーに保管するためのアダプターが提供されています。適切なアダプターを作成することにより、適当であればどんな場所あるいはメディア (DB2 データベースや CD-R など) にでも、メッセージを保管できるようになります。

メッセージの永続性や回復力は、どのアダプターを選択するかによって決まります。たとえば、メモリー・アダプターを使用する場合、メッセージの回復力はメモリーのそれ

と同じだけになります。メモリーは、ディスクよりもかなり高速なメディアですが、ディスクに比べて非常に不安定なメディアでもあります。それで、アダプターの選択が重要であるということが分かります。

キューの作成時にメッセージ・ストアが作成されない場合は、キュー・マネージャーを作成する際に指定されたデフォルトのメッセージ・ストアが使用されます。詳細については、45ページの『第4章 キュー・マネージャー、メッセージ、およびキュー』を参照してください。

Queue_FileDesc フィールドを設定する際には、以下の点を考慮に入れる必要があります。

- 使用している構文が、キューが置かれているシステムに対して適切なものであることを確認してください。たとえば、Windows システムではファイル区切りに "¥" を使用していますが、 UNIX® システムでは "/" を使用します。ある場合には、このどちらを使用しても構わないこともありますが、それは、キュー・マネージャーが稼働する JVM (Java 仮想マシン) のサポートによって異なります。また、ファイル区切りの違いに加え、Windows NT のような一部のシステムでは、UNIX などの他のシステムでは使用されないドライブ名を使用している場合があります。
- 一部のシステムでは、他のシステムでは使用できない、相対的なディレクトリー (例、".¥") を指定できる場合があります。相対的なディレクトリーを指定できるシステムにおいても、相対的なディレクトリーを指定する場合は厳重な注意が必要です。現行ディレクトリーは JVM の存続時間の間に変更される可能性があります。この種の変更は、相対的なディレクトリーを使用してキューの対話を行う際に、問題を引き起こします。

ローカル・キューの作成: 次のコード・フラグは、ローカル・キューをどのように作成するかを示したものです。

```
/**
 * Create a new local queue
 */
protected void createQueue(MQeQueueManager localQM,
                           String          qMgrName,
                           String          queueName,
                           String          description,
                           String          queueStore
  ) throws Exception
{
  /*
   * Create an empty queue admin message and parameters field
   */
  MQeQueueAdminMsg msg = new MQeQueueAdminMsg();
  MQeFields parms = new MQeFields();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );
```

```
  /*
   * Set name of queue to manage
   */
  msg.setName( qMgrName, queueName );

  /*
   * Add any characteristics of queue here, otherwise
   * charateristics will be left to default values.
   /
  if ( description != null ) // set the description ?
    parms.putUnicode( MQeQueueAdminMsg.Queue_Description,
                      description);

  if ( queueStore != null )  // Set the queue store ?
    // If queue store includes directory and file info then it
    // must be set to the correct style for the system that the
    // queue will reside on e.g ¥ or /
    parms.putAscii(MQeQueueAdminMsg.Queue_FileDesc,
                   queueStore );
  /*
   * Other queue characteristics like queue depth, message expiry
   * can be set here ...
   */

  /*
   * Set the admin action to create a new queue
   */
  msg.create( parms );

  /*
   *  Put the admin message to the admin queue (not assured delivery)
   */
  localQM.putMessage( qMgrName,
                      MQe.Admin_Queue_Name,
                      msg,
                      null,
                      0);
}
```

キューのセキュリティー: アクセスとセキュリティーはキューに属しており、リモート・キュー・マネージャーでの使用に関しては (ネットワークに接続する場合)、これら他のキュー・マネージャーがそのキューとメッセージをやり取りすることを許可する権限を設けることができます。キューのセキュリティーのセットアップでは、次の特性が使用されます。

- *Queue_Cryptor*
- *Queue_Authenticator*
- *Queue_Compressor*
- *Queue_TargetRegistry*
- *Queue_AttrRule*

セキュリティーに基づくキューのセットアップについての詳細は、221ページの『第8章 セキュリティー』を参照してください。

その他のキューの特性: キューを構成する際には、キューが受け取ることのできるメッセージの最大数などを含め、他の多くの特性を使用することができます。これらについては、「*MQSeries Everyplace for Multiplatforms プログラミング・リファレンス*」の *MQeQueueAdminMsg* のセクションを参照してください。

別名: キューの名前には、 138ページの『別名』で、接続に関連して説明されているものと同様の別名を付けることができます。接続のセクションにある別名の例のコード・フラグでは、接続に対して別名をセットアップする方法が示されています。キューに対して別名をセットアップする場合には、 MQeQueueAdminMsg の代わりに MQeConnectionAdminMsg を使用する点を除き、これと同じ方法を用いることができます。

アクションの制限: ある特定の管理アクションは、キューが事前に定義された状況にない場合には実行することができません。次のアクションがそうです。

Action_Update

- キューが使用されているときは、そのキューの特性を変更できません
- キューのセキュリティー特性は、キューにメッセージが含まれている場合には変更することができません。
- キューのメッセージ・ストアは、1 度設定されたなら変更できません

Action_Delete

キューは、使用されている場合、つまりそのキューにメッセージが含まれている場合には変更できません

キューが使用されていない、つまりメッセージがまったく含まれていない状況になければ実行できない要求は、キュー・マネージャーが再始動される際か一定の時間間隔の後に再試行されます。管理要求の再試行をセットアップすることについての詳細は、120ページの『基本となる管理要求メッセージ』を参照してください。

リモート・キュー

リモート・キューは MQeRemoteQueue クラスによってインプリメントされ、MQeAdminMsg のサブクラスである MQeRemoteQueueAdminMsg クラスによって管理されます。

キューの名前は、宛先キュー・マネージャーの名前 (リモート・キューの場合、これはそのキューがローカルに存在するキュー・マネージャーの名前) と、そのキュー・マネージャーでのキューの実名からとられます。キューは、管理メッセージ内の 2 つの ascii フィールド、すなわち *Admin_Name* フィールドと *Queue_QMgrName* フィールドによって一意的に識別されます。これら管理メッセージ内の 2 つのフィールドは、メソッド **setName(queueManagerName, queueName)** を使用して設定することができます。

リモート・キューの定義では、そのキューのキュー・マネージャー名と、定義が置かれているキュー・マネージャーの名前とが一致することはありません。

キューのリモート定義は、ほとんどの場合、実際のキューの定義と一致します。そうでない場合、そのキューと対話する際には異なった結果が得られます。たとえば、次のような場合があります。

- 非同期キューでは、リモート定義の *max message size* が実際のキューの同じ定義より大きい場合に、定義はリモート・キューのストレージで受け入れられますが、実際のキューに移動されると拒否されます。なおメッセージは、失われずにリモート・キューに残されます。ただし、これを送達することはできません。
- 同期キューでは、セキュリティー特性が一致しない場合に、MQSeries Everyplace は、実際のキューと折衝してどのセキュリティー特性を使用するべきかを決定します。メッセージの書き込みは正常に行われる場合もありますが、それ以外の場合には、属性の不一致による例外が返されます。

動作モードの設定: 同期的な操作のためにキューを設定するには、*Queue_Mode* フィールドを Queue_Synchronous に設定します。

非同期キューでメッセージを一時的に保管するには、メッセージ・ストアが必要です。メッセージ・ストアの定義は、ローカル・キューの場合と同じです (140ページの『メッセージ・ストア』を参照)。

非同期的な操作のためにキューを設定するには、 *Queue_Mode* フィールドを Queue_Asynchronous に設定します。

145ページの図20 は、同期的な操作のためのリモート・キューのセットアップと、非同期的な操作のためののりモード・キューのセットアップを示したものです。

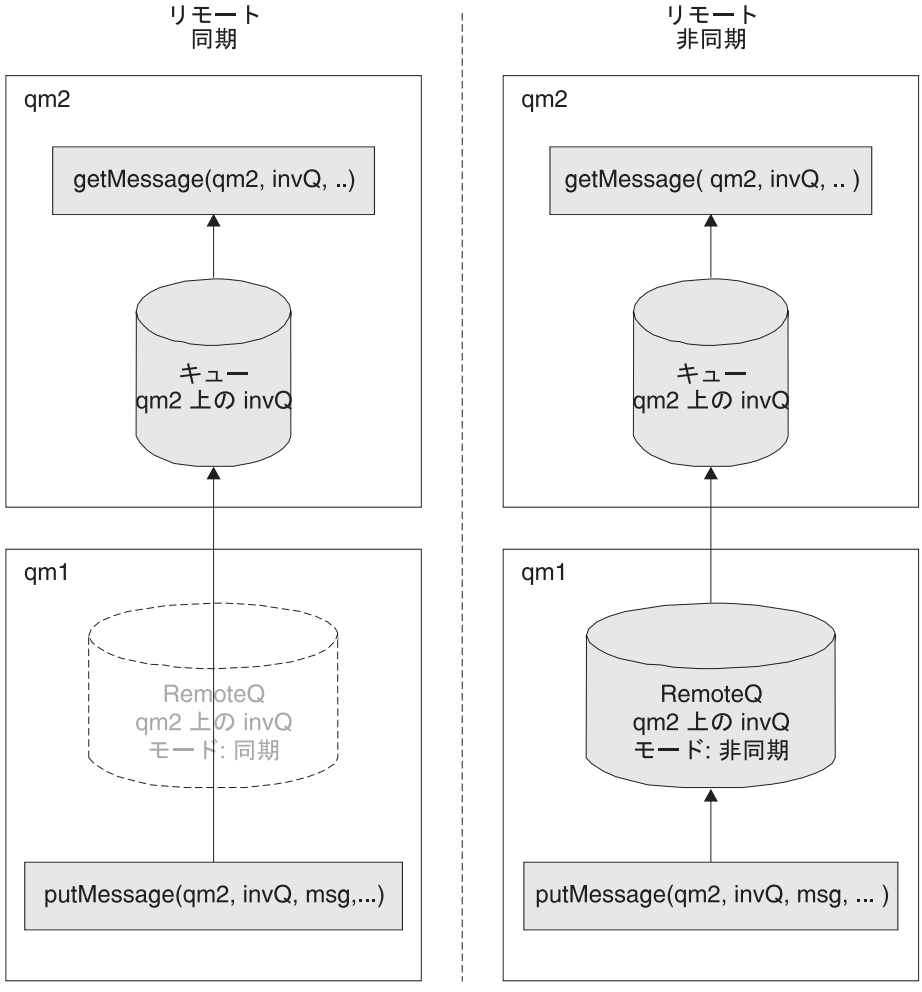

図 20. リモート・キュー

- 同期、非同期のいずれの例でも、キュー・マネージャー `qm2` には `invQ` というローカル・キューが存在します。
- 同期の例のキュー・マネージャー `qm1` には、キュー `invQ` のリモート・キュー定義があります。 `invQ` は、キュー・マネージャー `qm2` に常駐します。動作モードは同期に設定されています。

  キュー・マネージャー `qm1` を使用するアプリケーションと、キュー `qm2.invQ` への書き込みメッセージによって、キュー・マネージャー `qm2` に対するネットワーク接続が確立され (まだ接続が存在していない場合)、すぐにメッセージが実際のキューに書き込まれます。ネットワーク接続を確立できない場合、アプリケーションは例外を受け取り、これを処理しなければなりません。

- 非同期の例のキュー・マネージャー qm1 には、キュー invQ のリモート・キュー定義があります。 invQ は、キュー・マネージャー qm2 に常駐します。動作モードは非同期に設定されています。

  キュー・マネージャー qm1 を使用するアプリケーションと、キュー qm2.invQ への書き込みメッセージでは、qm1 のリモート・キューに一時的にメッセージを保管します。伝送の基準にかなうなら、メッセージはキュー・マネージャー qm2 上の実際のキューに移動されます。メッセージは、伝送が正常に行われるまでリモート・キューに保管されます。

リモート・キューの作成: 次のコード・フラグは、リモート・キューを作成するための管理メッセージをどのようにセットアップするかを示したものです。

```
/**
 * Create a remote queue
 */
protected void createQueue(MQeQueueManager localQM,
                           String          targetQMgr,
                           String          qMgrName,
                           String          queueName,
       String          description,
                           String          queueStore,
       byte            queueMode
  ) throws Exception
{
  /*
   * Create an empty queue admin message and parameters field
   */
  MQeRemoteQueueAdminMsg msg = new MQeRemoteQueueAdminMsg();
  MQeFields parms = new MQeFields();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of queue to manage
   */
  msg.setName( qMgrName, queueName );

  /*
   * Add any characteristics of queue here, otherwise
   * charateristics will be left to default values.
   /
  if ( description != null ) // set the description ?
    parms.putUnicode( MQeQueueAdminMsg.Queue_Description,
                      description);

  // set the queue access mode if mode is valid
  if ( queueStore != MQeQueueAdminMsg.Queue_Asynchronous &&
       queueStore != MQeQueueAdminMsg.Queue_Synchronous )
    throw new Exception ("Invalid queue store");
  parms.putByte( MQeQueueAdminMsg.Queue_Mode,
```

```
                queueMode);

  if ( queueStore != null )  // Set the queue store ?
    // If queue store includes directory and file info then it
    // must be set to the correct style for the system that the
    // queue will reside on e.g ¥ or /
    parms.putAscii( MQeQueueAdminMsg.Queue_FileDesc,
                    queueStore );
  /*
   * Other queue characteristics like queue depth, message expiry
   * can be set here ...
   */

  /*
   * Set the admin action to create a new queue
   */
  msg.create( parms );

  /*
   *  Put the admin message to the admin queue (not assured delivery)
   *  on the target queue manager
   */
  localQM.putMessage( targetQMgr,
                      MQe.Admin_Queue_Name,
                      msg,
                      null,
                      0);
}
```

非同期的な操作の場合、リモート・キュー定義に含めるキュー特性は、キュー・ディスカバリーを使用することによって取得できます。これについては、97ページで説明されています。

ストア・アンド・フォワード (蓄積交換) キュー

通常、このタイプのキューはサーバー上で定義され、次の 2 とおりに構成することができます。

- 次のキュー・マネージャーにメッセージを転送する。次のキュー・マネージャーは宛先キュー・マネージャーでなくても構いません。この場合は、ストア・アンド・フォワード (蓄積交換) キューがメッセージをネクスト・ホップにプッシュします。
- 宛先キュー・マネージャーがストア・アンド・フォワード (蓄積交換) キューからメッセージを収集できるようになるまでメッセージを保持する。このような構成は、ホーム・サーバー ・キューを使用することによって可能になります (151ページの『ホーム・サーバー・キュー』を参照してください)。このような構成を使用する場合、メッセージはストア・アンド・フォワード (蓄積交換) キューからプルされます。

ストア・アンド・フォワード (蓄積交換) キューは、 MQeStoreAndForwardQueue クラスによってインプリメントされます。これらは MQeRemoteQueueAdminMsg のサブクラスである MQeStoreAndForwardQueueAdminMsg クラスによって管理されます。サブクラスの主な追加機能としては、ストア・アンド・フォワード (蓄積交換) キューがメッセー

ジを保持できるキュー・マネージャーの名前を追加および除去することができます。キュー・マネージャー名は、 **Action_AddQueueManager** アクション、および **Action_RemoveQueueManager** アクションによって、追加したり削除したりすることができます。 1 つの管理メッセージで複数のキュー・マネージャー名を追加したり削除したりすることができます。 ascii 配置フィールドの *Queue_QMgrNameList* を設定すれば、メッセージの中に直接名前を書きこむことができます。 **addQueueManager()** メソッドや **removeQueueManager()** メソッドを使用することもできます。各メソッドはそれぞれ 1 つのキュー・マネージャー名しか扱えませんが、メソッドを繰り返し呼び出して 1 つのメッセージに複数のキュー・マネージャー名を追加できます。

次のコード・フラグは、どのようにしてメッセージに宛先キュー・マネージャーの名前を追加するかを示しています。

```
/**
 * Setup an admin msg to add target queue managers to
 * a store and forward queue.
 */
public MQeStoreAndForwardQueueAdminMsg addQueueManager( String queueName
                                                         String queueManagerName
                                                         String qMgrNames[] )
                                                         throws Exception
{
  /*
   * Create an empty admin message
   */
  MQeStoreAndForwardQueueAdminMsg msg =
  new MQeStoreAndForwardQueueAdminMsg();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of the store and forward queue
   */
  msg.setName( queueManagerName, queueName );
  /*
   * Use the addAlias method to add aliases to the message.
   */
  for ( int i=0; i<qMgrNames.length; i++ )
  {
    msg.addQueueManager(qMgrNames[i] );
  }

  return msg;
}
```

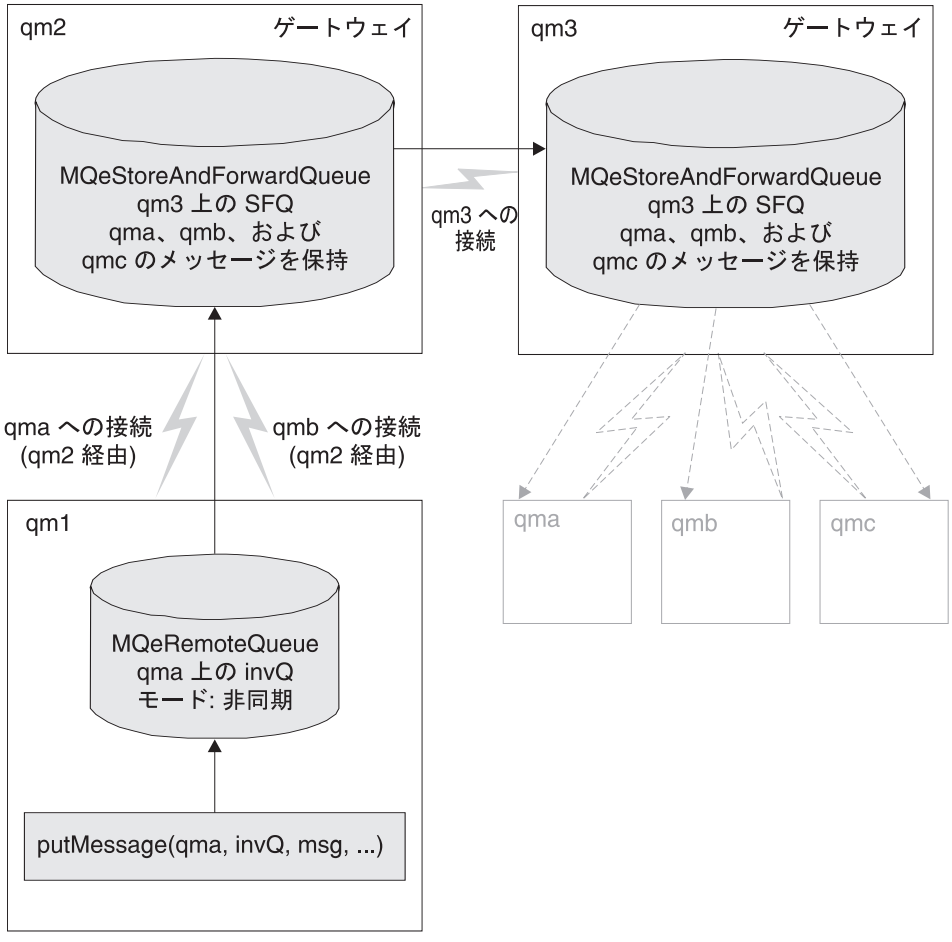

図 21. ストア・アンド・フォワード (蓄積交換) キュー

ストア・アンド・フォワード (蓄積交換) キューでメッセージを次のキュー・マネージャーへプッシュするには、ストア・アンド・フォワード (蓄積交換) キューのキュー・マネージャー名属性は、次のキュー・マネージャーの名前でなければなりません。また、次のキュー・マネージャーへの接続を構成する必要もあります。

また、メッセージの収集 (プル) を待つようにストア・アンド・フォワード (蓄積交換) キューを構成する場合には、ストア・アンド・フォワード (蓄積交換) キューのキュー・マネージャー名属性は意味を持ちません。キュー名のキュー・マネージャー属性に関する唯一の制限は、同じ名前で接続を行うことできないという点です。このような接続がある場合、キューは、メッセージの転送にその接続を使用することを試みます。

149ページの図21 は、別々のキュー・マネージャーにある 2 つのストア・アンド・フォワード (蓄積交換) キューの例を示しています。 1 つは次のキュー・マネージャーにメッセージをプッシュするようセットアップされたキューで、他方はメッセージが収集されるのを待つようにセットアップされています。

- キュー・マネージャー qm2 では、キュー・マネージャー qm3 に対する接続が構成されています。
- キュー・マネージャー qm2 では、接続 qm3 を使用してキュー・マネージャー qm3 にメッセージをプッシュするよう構成されたストア・アンド・フォワード (蓄積交換) キューがあります。ストア・アンド・フォワード (蓄積交換) キューのキュー・マネージャー名属性は、接続名と同じ qm3 です。
- qm2 上のストア・アンド・フォワード (蓄積交換) キュー qm3.SFQ は、キュー・マネージャー qma、qmb、および qmc を宛先とするメッセージを扱うように構成されています。
- キュー・マネージャー qm3 には、ストア・アンド・フォワード (蓄積交換) キュー qm3.SFQ があります。キュー名 qm3 のキュー・マネージャー名部分には、これに対応する qm3 という接続がありません。したがってすべてのメッセージは、収集されるまでキューに保管されます。
- qm3 のストア・アンド・フォワード (蓄積交換) キュー qm3.SFQ には、キュー・マネージャー qma、qmb、および qmc に代わってメッセージが保管されます。メッセージは、収集されるか有効期限が切れるまで保管されます。

キュー・マネージャーが、中間キュー・マネージャーにあるストア・アンド・フォワード (蓄積交換) キューを使用して他のキュー・マネージャーにメッセージを送ろうとする場合、最初のキュー・マネージャーには以下のものが必要です。

- 中間キュー・マネージャーに対して構成された接続
- 中間キュー・マネージャーを介し、宛先キュー・マネージャーに対して構成された接続
- 宛先キューのリモート・キュー定義

これらの条件がすべて満たされていれば、アプリケーションは、キュー・マネージャー・ネットワークのレイアウトを理解していなくても、宛先キュー・マネージャーの宛先キューにメッセージを書き込むことができます。これは、基礎となるキュー・マネージャー・ネットワークに変更を加えても、アプリケーション・プログラムには影響しないということを意味します。

149ページの図21 の中で、キュー・マネージャー qm1 は、キュー・マネージャー qma のキュー invQ にメッセージを書き込めるよう構成されれています。この構成は、次のものから成っています。

- 中間キュー・マネージャー qm2 への接続
- 宛先キュー・マネージャー qma への接続
- qma 上のリモート非同期キュー invQ

アプリケーションが、キュー・マネージャー qm1 を使用してキュー・マネージャー qma のキュー invQ にメッセージを書き込む場合、メッセージの流れは次のようになります。

1. アプリケーションが非同期キュー qma.invQ にメッセージを書き込みます。メッセージは、伝送の基準に従ってネクスト・ホップに移されるまで、qm1 にローカルに保管されます。
2. 伝送の基準にかなえば、メッセージが移されます。 qma の接続の定義に基づいて、メッセージはキュー・マネージャー qm2 を経由します。
3. キュー・マネージャー qma のキュー invQ に対するメッセージを扱うように構成されているキューは、 qm2 のストア・アンド・フォワード (蓄積交換) キュー qm3.SFQ だけです。メッセージはこのキューに一時的に保管されます。
4. このストア・アンド・フォワード (蓄積交換) キューには、ネクスト・ホップ、つまりキュー・マネージャー qm3 にメッセージをプッシュするための接続が構成されています。
5. キュー・マネージャー qm3 には、キュー・マネージャー qma を宛先としたメッセージを保持できるストア・アンド・フォワード (蓄積交換) キュー qm3.SFQ があり、メッセージはこのキューに保管されます。
6. qma へのメッセージは、キュー・マネージャー qma によって収集されるまで、ストア・アンド・フォワード (蓄積交換) キューで保管されます。このセットアップを行う方法については、『ホーム・サーバー・キュー』を参照してください。

ホーム・サーバー・キュー

ホーム・サーバー・キューは、 MQeHomeServerQueue クラスによってインプリメントされます。これらは MQeRemoteQueueAdminMsg のサブクラスである MQeHomeServerQueueAdminMsg クラスによって管理されます。サブクラスに追加されているのは、 *Queue_QTimerInterval* 特性だけです。このフィールドは int タイプのフィールドで、ここにはミリ秒単位の時間間隔が設定されます。このフィールドに 0 以上の値が設定されると、ホーム・サーバー・キューは、ホーム・サーバーを n ミリ秒ごとに検査して、収集を待っているキューがないかどうかを確認します。待機中のメッセージは、すべて宛先キューに送達されます。 このフィールドの値が 0 であれば、ホーム・サーバーのポーリングが行われるのは、**MQeQueueManager.triggertransmission** メソッドが呼び出されたときのみであることを意味します。

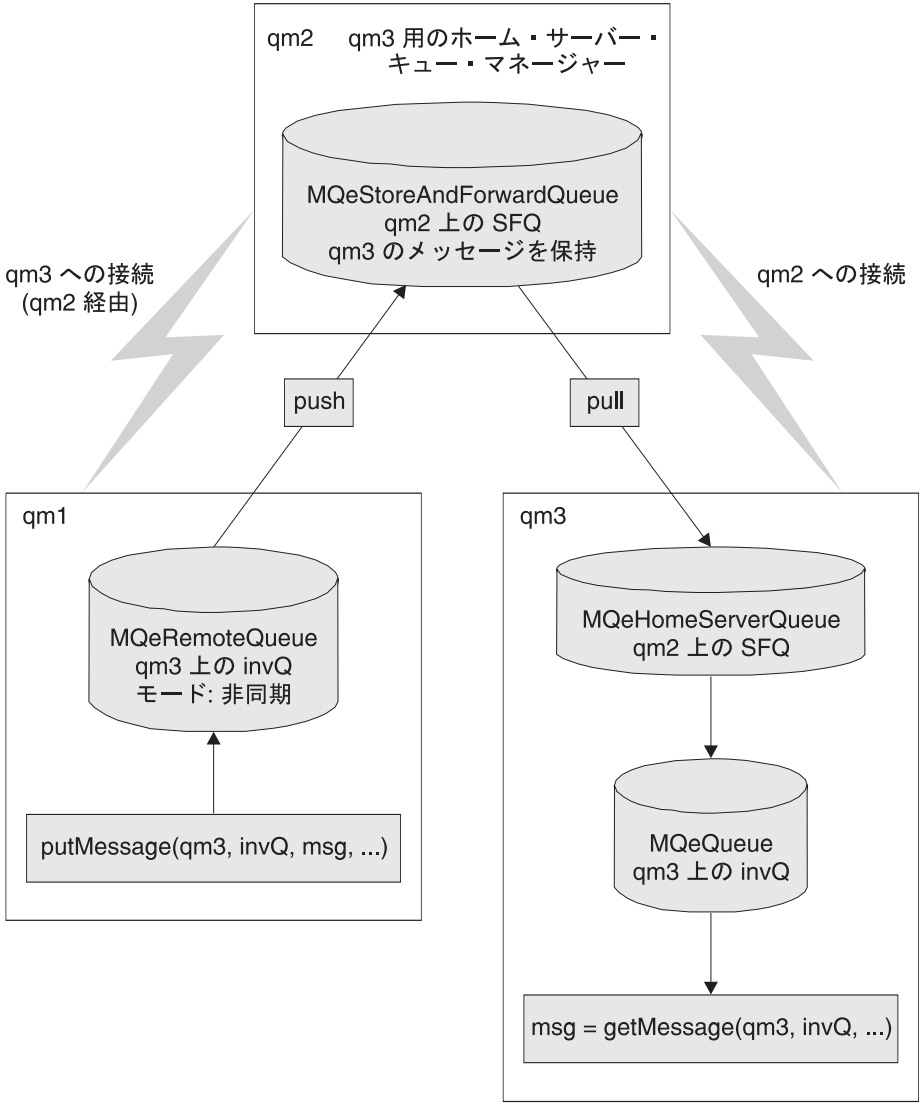

図 22. ホーム・サーバー・キュー

ホーム・サーバー・キューの名前は次のように設定します。

- キュー名はストア・アンド・フォワード (蓄積交換) キューの名前と一致していなければならない
- キュー名のキュー・マネージャー属性はホーム・サーバー・キュー・マネージャーの名前と一致していなければならない

ホーム・サーバー・キューがあるキュー・マネージャーには、ホーム・サーバー・キュー・マネージャーに対して構成された接続が必要です。

152ページの図22 は、ホーム・サーバー・キュー SFQ を持つキュー・マネージャー qm3 の例を示しています。このキューは、ホーム・サーバー・キュー・マネージャー qm2 からメッセージを収集するように構成されています。

この構成は、次のものから成っています。

- ホーム・サーバー・キュー・マネージャー qm2
- キュー・マネージャー qm3 へのメッセージを保持する、キュー・マネージャー qm2 のストア・アンド・フォワード (蓄積交換) キュー SFQ
- 通常はキュー・マネージャー qm2 から切断された状態で稼働し、またそれから接続を受けることができないキュー・マネージャー qm3
- キュー・マネージャー qm3 から qm2 に対して構成された接続
- ホーム・サーバーとしてキュー・マネージャー qm2 を使用するホーム・サーバー・キュー SFQ

qm2 を通してキュー・マネージャー qm3 に送信されるすべてのメッセージは、qm3 のホーム・サーバー・キューによって収集されるまで、qm2 のストア・アンド・フォワード (蓄積交換) キュー SFQ に保管されます。

MQSeries-ブリッジ キュー

MQSeries-ブリッジ キューとは、MQSeries キュー・マネージャーにあるキューを参照する、リモート・キュー定義のことをいいます。メッセージを保持するキューは、ローカル・キュー・マネージャーではなく、MQSeries キュー・マネージャーにあります。

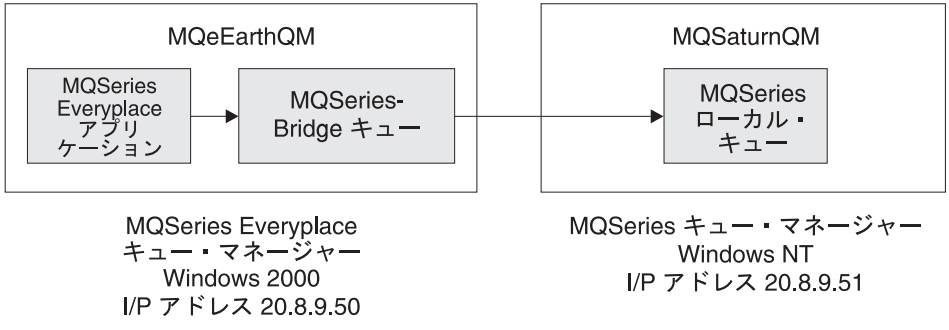

図 23. MQSeries-ブリッジ キュー

- MQSaturnQM MQSeries キュー・マネージャーには、ローカル・キュー MQSaturnQ が定義されています。
- MQeEarthQM では、 MQSaturnQM キュー・マネージャー上に MQSaturnQ という MQSeries-ブリッジ・キューを定義する必要があります。

- MQeEarthQM キュー・マネージャーに付加されたアプリケーションは MQSaturnQ MQSeries-ブリッジ・キューにメッセージを書き込み、ブリッジ・キューは MQSaturnQM キュー・マネージャーの MQSaturnQ にそのメッセージを送達します。

ブリッジ・キューの定義では、ブリッジ・オブジェクト階層でクライアント接続を固有に識別するため、ブリッジ (MQSeries キュー・マネージャー・プロキシー) とクライアント接続の名前を指定する必要があります (175ページの図33 を参照してください)。この情報は、MQSeries-ブリッジが MQSeries キューを操作するために MQSeries キュー・マネージャーにアクセスする方法を識別します。

MQSeries-ブリッジ には、直接 MQSeries-ブリッジに接続していないキュー・マネージャー上のキューにメッセージを書きこむための機能があります。これにより、他の MQSeries キュー・マネージャーを介して MQSeries キュー・マネージャー (宛先) にメッセージを送信することが可能になります。 MQSeries-ブリッジ キューは、宛先キュー・マネージャーの名前を取り、中間キュー・マネージャーの名前は、 MQSeries キュー・マネージャー・プロキシーによって決まります。

MQSeries-ブリッジ キューで使用されるすべての特性のリストは、「*MQSeries Everyplace for Multiplatforms プログラミング・リファレンス* 」の *com.ibm.mqe.bridge* セクションの *MQeMQBridgeQueueAdminMsg* を参照してください。

表11 は、 MQSeries-ブリッジ キューが構成されるとサポートされるようになる操作の詳細リストです。

表 11. MQSeries-ブリッジ・キューでサポートされているメッセージ操作

操作のタイプ	**MQSeries-ブリッジ キューでのサポート**
getMessage()	なし
putMessage()	あり
browseMessage()	なし
browseAndLockMessage	なし

サポートされていない操作の使用がアプリケーションで試行された場合は、Except_NotSupported の MQeException が返されます。

アプリケーションがブリッジ・キューにメッセージを書き込むと、ブリッジ・キューは、そのブリッジのクライアント接続オブジェクトで保守されている接続のプールから、MQSeries キュー・マネージャーに対する論理接続を使用します。MQSeries への論理接続は、 MQSeries Java バインディング・クラスか MQSeries Java クラスでサポートされます。クラスの選択は、 MQSeries キュー・マネージャー・プロキシーの設定で **ホスト名** のフィールドに設定された値に従って行われます。MQSeries-ブリッジ・キューは、MQSeries キュー・マネージャーに接続されると、その MQSeries キューに対してメッセージの書き込みを試行します。

MQSeries-ブリッジ・ブリッジ・キューのアクセス・モードは、常に同期でなければならず、非同期キューとして構成することはできません。これは、書き込み操作が直接 MQSeries-ブリッジ・キューを操作し、正常に完了すれば、メッセージは、プロセスが書き込み操作の完了を待っている間に MQSeries システムに渡されているということを意味します。

MQSeries-ブリッジ・キューに対して同期操作を使用することをアプリケーションで望まない場合は、 MQSeries-ブリッジ・キューを参照する非同期のリモート・キュー定義をセットアップ (96ページの『非同期メッセージング』を参照) するか、または、その代わりとしてストア・アンド・フォワード (蓄積交換) キューとホーム・サーバー・キューをセットアップすることができます。後者の 2 つの構成を代用するなら、アプリケーションから非同期キューにメッセージを書き込むことができます。この構成で **putMessage()** メソッドが返された場合は、必ずしもメッセージが MQSeries キュー・マネージャーに渡されているとは限らない場合があります。

MQSeries-ブリッジ・キューの使用の例は、179ページの『構成の例』で取り上げられています。

管理キュー

管理キューは、クラス MQeAdminQueue でインプリメントされる MQeQueue のサブクラスであるため、ローカル・キューと同じ機能を持っています。このキューは、管理クラス MQeAdminQueueAdminMsg を使用して管理されます。

管理されるリソースが使用されているためにメッセージが失敗した場合は、そのメッセージを再試行するように要求することができます。試行回数の最大値のセットアップについては、120ページの『基本となる管理要求メッセージ』で詳細に説明しています。管理対象リソースが使用可能でないためにメッセージが失敗し、試行回数がまだ最大値に達していない場合は、後日処理するためにメッセージはキューに残されます。試行回数が最大値に達すると、その要求は MQeException で失敗します。デフォルトでは、次にキュー・マネージャーが再始動される際にメッセージが再試行されますが、キューにタイマーを設定し、指定した間隔の後にキューでメッセージを処理することもできます。タイマーの間隔は、管理メッセージの長フィールド *Queue_QTimerInterval* フィールドで指定することができます。間隔値はミリ秒単位で指定されます。

セキュリティーと管理

デフォルトでは、すべての MQSeries Everyplace アプリケーションが管理対象リソースを管理することができます。アプリケーションは、管理されるキュー・マネージャーに対してローカル・アプリケーションとして稼働することもできますし、別のキュー・マネージャーで稼働することもできます。管理アクションの保護は重要です。セキュリティーが保護されなければ、システムは誤用される恐れがあります。 MQSeries Everyplace

には、管理のセキュリティーを保護するための基本的な機能が備わっています。これに使用されるキューのセキュリティーについては、221ページの『第8章 セキュリティー』を参照してください。

非同期セキュリティーを使用する場合は、管理キューにセキュリティー特性を設定することによって、そのセキュリティーを保護することができます。たとえば、ユーザーが管理アクションを実行するには、その前に、認証を設定することによって、そのユーザーにオペレーティング・システム (Windows NT または UNIX) への認証を与える必要があります。これを拡張して、特定のユーザーだけが管理を行えるようにすることができます。

管理キューでは、アプリケーションが直接これにアクセスしてメッセージを書き込むことは許されておらず、メッセージは内部的に処理されます。これはつまり、メッセージ・レベルのセキュリティーによって保護されているキューに書き込まれたメッセージは、読み取りやブラウズの要求において属性を提供する通常のメカニズムでは開くことができないということを意味します。そのようなメッセージを管理キューで処理するためには、まず、キュー・ルール・クラスを管理キューに適用して、保護されているすべてのメッセージを開くことができます。メッセージを開いて管理を行えるようにするためには、キュー・ルール、**browseMessage()** をコード化する必要があります。

キュー・ルールのインプリメントに関する追加情報は、115ページの『キュー・ルール』を参照してください。

管理コンソールの例

MQSeries Everyplace で提供されている例の 1 つは、管理のグラフィカル・ユーザー・インターフェース (GUI) です。この例では、本書の前の方のセクションで扱われた多くの管理技法や機能を利用します。この例のすべてのクラスはパッケージ examples.administration.console に含まれています。

この例では、次のような MQSeries Everyplace 管理機能を示します。

- ローカル・キュー・マネージャーとリモート・キュー・マネージャーの両方の管理
- すべての MQSeries Everyplace 管理対象リソースの管理
- 各管理対象リソースのすべてのアクションに対するアクセス
- ほとんどの基本的な MQeAdminMsg 機能の使用
- キュー・ブラウザー
- 管理応答キューのキュー・ブラウザーのカスタマイズ・バージョン

ここで扱われる内容はすべて、プログラミングの例として紹介されており、***開発およびテスト環境外で使用するようには意図されていません。***この例の機能は、トレースやクライアント・キュー・マネージャーといった他の例と関連して使用されていること、お

よび、MQSeries-ブリッジ の管理の例を示すためにサブクラス化されたものであることにご注意ください (187ページの『管理 GUI アプリケーションの例』を参照)。

メイン・コンソール・ウィンドウ

コンソールを開始するには、次のコマンドを使用します。

```
java examples.administration.console.Admin
```

これにより、次のウィンドウが表示されます。

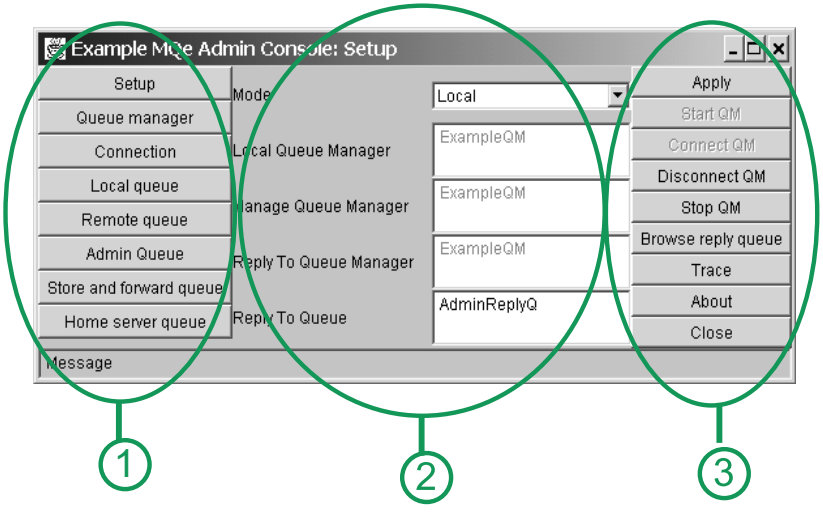

図 24. 管理コンソール・ウィンドウ

これは中心となるウィンドウで、他のすべての対話はこのウィンドウから開始されます。ウィンドウは、次のような 3 つのセクションに分かれています。

1. 管理するリソースのタイプ

ウィンドウの左側にある一連のボタンは、管理するリソースの選択を制御します。ここには、MQSeries Everyplace 管理対象リソースのタイプごとのボタンが 1 つずつと、**「セットアップ (Setup)」**という特別なボタンが 1 つあります。**「セットアップ (Setup)」**ボタンからは、応答先キューのブラウズやトレースのオン / オフといった、一連の基本的な管理機能にアクセスすることができます。

2. 基本的な管理のパラメーター

ウィンドウの中央部分では、基本的な管理のパラメーターを調整することができます。

「モード (Mode)」:

管理するキュー・マネージャーがローカルであるかリモートであるか。

「ローカル・キュー・マネージャー *(Local queue manager)*」:
管理アクションを開始するローカル・キュー・マネージャーの名前。**「QM の開始 (Start QM)」**ボタンでキュー・マネージャーが開始された場合、これは自動的に設定されます。

「リモート・キュー・マネージャー *(Remote queue manager)*」:
モードがリモートに設定されている場合は、これが管理するキュー・マネージャーの名前です。モードがローカルに設定されている場合、これは、常にローカル・キュー・マネージャーと同じになります。

「応答先キュー・マネージャー *(Reply-to queue manager)*」:
管理応答メッセージが送られるキュー・マネージャーの名前。

「応答先キュー *(Reply-to queue)*」:
管理応答メッセージが送られるキューの名前。

3. 管理対象リソースに固有のアクション
各管理対象リソースには、そこで実行できる一連のアクションというものが決まっています。メイン・ウィンドウの右側に表示される一連のボタンは、ウィンドウ左側で選択されるリソースに使用できるアクションを示しています。いずれかのアクションのボタンを選択すると、そのアクションの機能が開始します。 (通常はそのアクションに関連する別のウィンドウが表示されます)。

選択されたローカル・キュー・マネージャーは、コンソールを実行している JVM の中で実行している必要があります。キュー・マネージャーがまだ実行していない場合は、**「QM の開始 (Start QM)」**ボタンを使用して開始する必要があります。そうすると、キュー・マネージャーの始動パラメーターの入った ini ファイルの名前とパスを要求するダイアログが表示されます。キュー・マネージャーがすでに実行している場合は、**「QM に接続 (Connect QM)」**ボタンを選択することができます (これは、管理が例示サーバー ExampleAwtMQeServer から開始されている場合です)。

キュー・マネージャーが開始されたなら、任意のリソースを選択および管理することができます。

キュー・ブラウザー

キュー・ブラウザーの例として、MQSeries Everyplace には、 AdminQueueBrowser が提供されています。ここでは、この例を通して、キューをブラウズし、キューにあるメッセージの内容を表示する方法を紹介します。ブラウズできるのは、同期的にアクセスでき、かつユーザーがアクセスに必要な権限を持っているキューだけです。メッセージ・レベルのセキュリティーを使用して保護されているメッセージを、サンプル・コードを使用して表示することはできません。

AdminQueueBrowser は、拡張機能を使用して管理応答キューをブラウズすることができるようにサブクラス化されています。これは、クラス AdminLogBrowser にインプリメ

ントされます。サブクラスにアクセスするには、**「セットアップ (Setup)」**ボタンを選択し、その次に**「応答キューのブラウズ (Browse reply queue)」**ボタンを押します。

次の図は、管理応答キュー・ウィンドウの概観を示したものです。

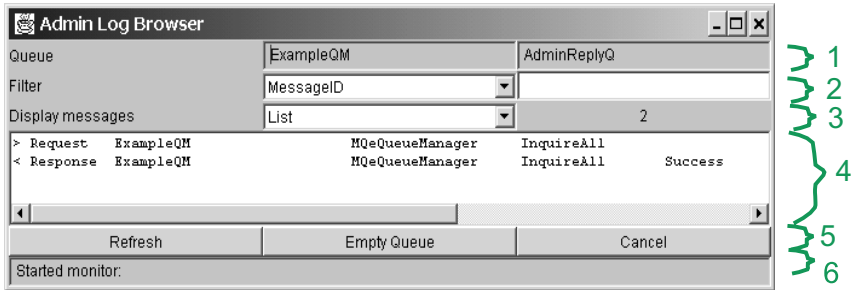

図 25. 応答先キュー・ウィンドウ

ウィンドウは、次のようないくつかのセクションに分かれています。

1. キュー・マネージャーとキューに対する管理応答の名前

2. メッセージ・フィルター

フィルターを使用して、表示するメッセージのセットに制限を設けることができます。この例では、メッセージの *MsgID* フィールドと *CorrelID* フィールドでフィルターを行うことができます。また例では、バイト配列でエンコードされたストリングを含むフィールドを表示させるという前提も設けています。

管理メッセージが例のコンソールから送信されると、 *MsgID* は管理されるキュー・マネージャーの名前に設定されます。これにより、特定のキュー・マネージャーに対する管理メッセージだけを表示させることが可能になります。

3. メッセージ表示のタイプ

メッセージ表示パネル内のメッセージは、次のような方法で表示することができます。

「リスト (List)」:

1 行に要約されたキュー内の各メッセージをリストします。

「全表示 (Full)」:

キュー上のすべてのメッセージの内容を表示します。

「リストと全表示 (Both)」:

2 つのパネルを使用し、1 方には各メッセージを要約する行のリストを、他方にはメッセージ・パネルで選択されたメッセージの内容を表示します。

現在見ているメッセージの数も表示されます。

4. メッセージ表示パネル

3 でも説明した通り、このパネルにはさまざまな形式でメッセージが表示されます。メッセージの詳細表示を新しいウィンドウに表示させるには、表示リストの中のそのメッセージをダブルクリックします。

5. アクション

次のようないくつかのボタンを通して、そのキュー・ブラウザーに固有のアクションを使用できます。

「最新表示 (Refresh)」

画面をクリアし、キューの最新の内容を表示します。ブラウズしているキューがローカル・キューである場合は、自動的にモニターが開始されます。このモニターは、キューに新しいメッセージが追加されると画面を最新表示します。ブラウズされているキューがリモート・キューである場合は、新しいメッセージが追加されても自動的にウィンドウを最新表示することはできません。この場合には、**「最新表示 (Refresh)」**ボタンを使用してキューの最新の内容を表示することができます。

「キューのクリア (Empty Queue)」

キューからすべてのメッセージを削除します。

「取消 (Cancel)」

キュー・ブラウザーのウィンドウをクローズします。

6. メッセージ

ここには、エラー・メッセージや状況メッセージが表示されます。

アクション・ウィンドウ

管理するリソースのタイプを選択し、アクションのボタンを選択すると、ウィンドウがオープンされ、そのアクションで使用できるパラメーターのリストが表示されます。これらのパラメーターには、必須のものとオプションとして使用されるものがあります。次の図は、接続における追加アクションの選択の例を示しています。

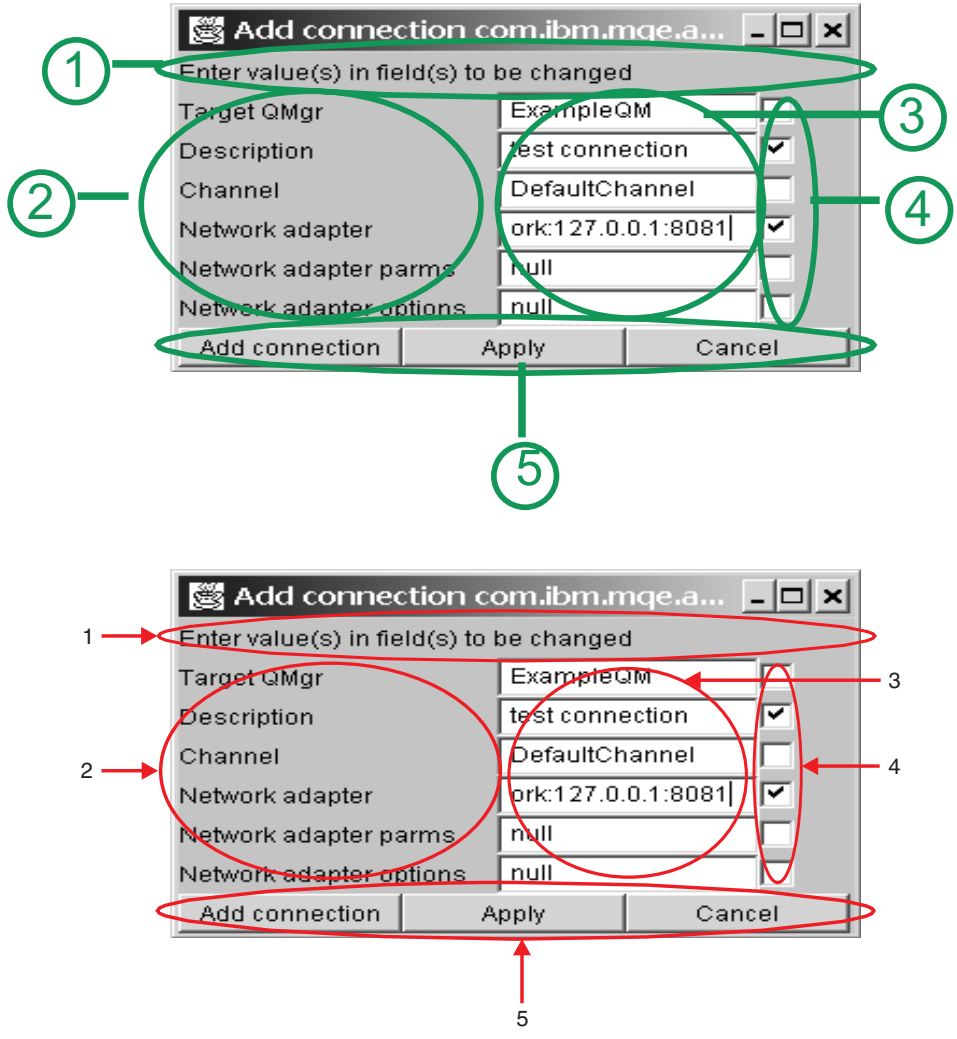

図 26. アクション・ウィンドウ

アクション・ウィンドウは、ほとんどのアクションで共通しています。このウィンドウは、次の部分から成っています。

1. メッセージ領域
ここには、エラー・メッセージや状況メッセージが表示されます。

2. パラメーターの名前
アクション・パラメーター名。

3. パラメーターの値
入力フィールド。パラメーターの値はここで変更することができます。表示される初期値は、そのパラメーターのデフォルト値です。

4. 送信フィールド

各フィールドでは、値が変更されると自動的にこのチェック・ボックスが選択されます。このフィールド (チェック・ボックス) が選択されていると、そのフィールドは管理メッセージに組み込まれます。デフォルトでは、変更されている値だけが管理メッセージに組み込まれ、デフォルトの値は組み込まれません。管理メッセージは、メッセージのサイズを最小限に保つために、デフォルトの値を認識してこれをメッセージに含めません。なお、値をデフォルトに戻す場合は、手動で送信フィールドのチェック・ボックスを選択する必要があります。

5. アクション・ボタン

各管理アクションごとに、次の 3 つのボタンがあります。

「アクション (Action)」

このボタンの名前は、管理アクションによって異なります (この例では**「接続の追加 (Add connection)」**)。アクションが実行されるときは、いつでも管理メッセージが作成され、それが宛先キュー・マネージャーに送信されます。アクション・ウィンドウはクローズします。

「適用 (Apply)」

管理メッセージを作成し、それを宛先キュー・マネージャーに送信します。アクション・ウィンドウはオープンされたまま残り、同じメッセージを複数回送信したり、メッセージを変更して送信することができます。

「取消 (Cancel)」

管理メッセージを送信せずにアクション・ウィンドウをクローズします。

応答ウィンドウ

管理要求の結果は、 158ページの『キュー・ブラウザー』で説明した通り、管理ログ・ブラウザーを使って見ることができます。要求の結果について詳細を表示する場合には、リスト表示されている中から応答メッセージをダブルクリックします。

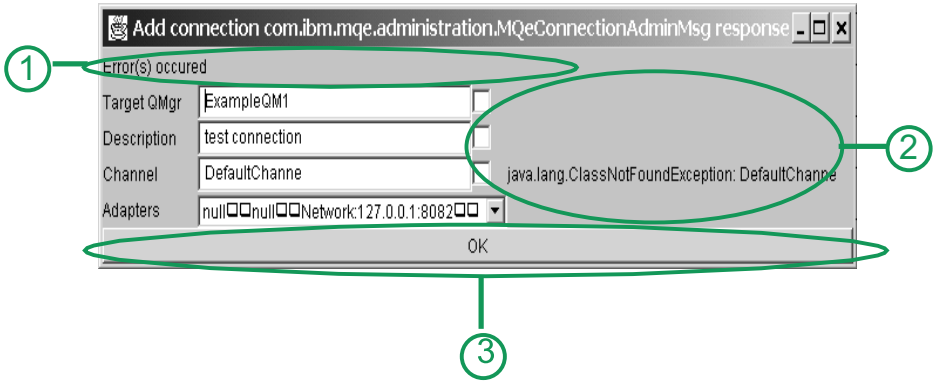

図 27. 応答ウィンドウ

このウィンドウの基本的な構造は管理要求のアクション・ウィンドウと同じですが、次の点が異なっています。

1. メッセージ

戻りコードとアクションの結果が表示されます。

2. エラーの詳細

戻りコードが `RC_Mixed` である場合は、特定のフィールドに関連するすべてのエラーがそのフィールドの隣りに表示されます。

3. アクション・ボタン

「OK」

アクション応答ウィンドウをクローズします。

コマンド行からの管理

MQSeries Everyplace には、簡単なスクリプトを使用して MQSeries Everyplace オブジェクトをコマンド行から管理できるようにするいくつかのツールが組み込まれています。以下のようなツールが提供されます。

QueueManagerUpdater

ini ファイルからデバイス・キュー・マネージャーを作成し、キュー・マネージャーの特性を更新するための管理メッセージを送信します。

IniFileCreator

クライアント・キュー・マネージャーに必要なコンテンツを持った ini ファイルを作成します。

LocalQueueCreator

クライアント・キュー・マネージャーを開き、ローカル・キュー定義をそれに追加し、キュー・マネージャーを閉じます。

HomeServerCreator
サーバー・キュー・マネージャーを開き、ホーム・サーバー・キューを追加し、キュー・マネージャーを閉じます。

ConnectionCreator
Java プログラミングをしなくても、MQSeries Everyplace キュー・マネージャーに接続を追加できるようにします。

RemoteQueueCreator
デバイス・キュー・マネージャーを開いて使用し、それに管理メッセージを送信して、リモート・キュー定義が作成できるようにしてから、キュー・マネージャーを閉じます。

MQBridgeCreator
MQSeries Everyplace キュー・マネージャー上に MQSeries-ブリッジ を作成します。

MQQMgrProxyCreator
ブリッジ用の MQSeries キュー・マネージャーを作成します。

MQConnectionCreator
プロキシー・オブジェクトに関する MQSeries システム用の接続定義を作成します。

MQListenerCreator
MQSeries からメッセージを引き出すために MQSeries 伝送キュー・リスナーを作成します。

MQBridgeQueueCreator
MQSeries キューのメッセージを参照できる MQSeries Everyplace キューを作成します。

StoreAndForwardQueueCreator
ストア・アンド・フォワード・キューを作成します。

StoreAndForwardQueueQMgrAdder
ストア・アンド・フォワード・キューがメッセージを受け入れるために使用するキュー・マネージャー名を、キュー・マネージャーのリストに追加します。

以下のファイルも提供されます。

サンプル・スクリプト・ファイル
2 つの .bat サンプル・ファイル、および、ブランチ、ゲートウェイ、および MQSeries システムを含め、架空のネットワーク構成のセットアップを示すための runmqsc スクリプト。

Rolled-up Java example
バッチ言語独立性のために、どのようにしてバッチ・ファイルを Java ファイルにロールアップできるかを示す例。

コマンド行ツールの使用例

コマンド行ツールでは、スクリプトを使用して初期キュー・マネージャー構成を作成することができ、しかも Java プログラム言語によるプログラミング方法を知っていなくてもかまいません。

以下の例は、これらのツールを使用して図のようなネットワーク・トポロジーを構成する方法を示しています。

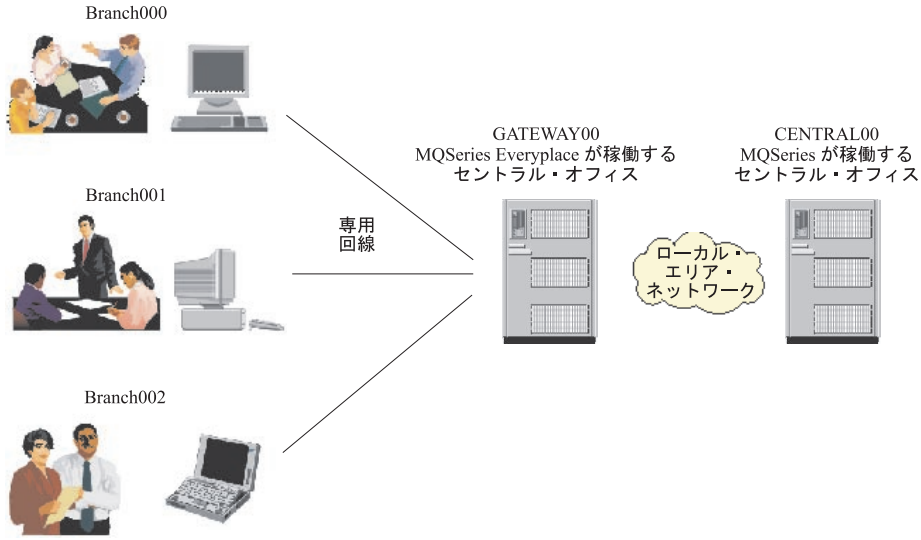

図 28. MQSeries Everyplace 管理シナリオ

このシナリオの場合:

- これらの事業所は、販売情報を中央側へ送信する必要があります。販売情報は、MQSeries サーバーのアプリケーションによって処理されます。
- 各事業所は 1 つのマシンを持っていて、DNS 名はそれぞれ `BRANCH000`、`BRANCH001`、および `BRANCH002` です。これらのマシンは、すべて MQSeries Everyplace を実行し、それぞれ `BRANCH000QM`、`BRANCH001QM`、および `BRANCH002QM` という 1 つのキュー・マネージャーを持っています。
- 本社のマシン `GATEWAY00` は、 1 つの ゲートウェイ・キュー・マネージャー `GATEWAY00QM` を実行します。
- 本社のマシン `CENTRAL00` は、 1 つのキュー・マネージャー `CENTRAL00QM` を持つ MQSeries を実行します。
- 販売が行われると、メッセージが MQSeries キュー・マネージャー `CENTRAL00QM` に送信され、 `BRANCH.SALES.QUEUE` というキューに入れられます。

- これらのメッセージは、事業所でバイト配列にエンコードされ、MQeMQMsgObject 内に送信されます。
- MQSeries システムは、メッセージを各ブランチ・キュー・マネージャーに返送できなければなりません。
- トポロジーも、後で事業所と ゲートウェイ 間のファイアウォールの追加に対応できるようになっていなければなりません。
- MQSeries 向けのキュー・トラフィックは、56 ビットの DES 暗号を使用しなければなりません。

必須スクリプト・ファイル

このネットワーク・トポロジーを構成するには、以下のスクリプトが必要です。

Central.tst
: CENTRAL00QM で関連オブジェクトを作成するために runmqsc スクリプトで使用します。

CentralQMDetails.bat
: CENTRAL00QM を他のスクリプトに記述するために使用します。

GatewayQMDetails.bat
: GATEWAY00QM を他のスクリプトに記述するために使用します。

CreateGatewayQM.bat
: ゲートウェイ・キュー・マネージャーを作成するために使用します。

CreateBranchQM.bat
: ブランチ・キュー・マネージャーを作成するために使用します。

これらの .bat ファイルはすべて、インストール済み製品の MQe\Java\Demo\Windows に入っています。

注: 提供されるサンプル・スクリプトは Windows の .bat ファイル・フォーマットになっていますが、それらはユーザーのシステムで使用できる任意のスクリプト言語で適切に機能するように変換できます。

スクリプトによって定義された MQSeries Everyplace および MQSeries オブジェクト

以下のオブジェクトは、事業所 - 本社経路指定を提供するためにスクリプトによって作成されます。

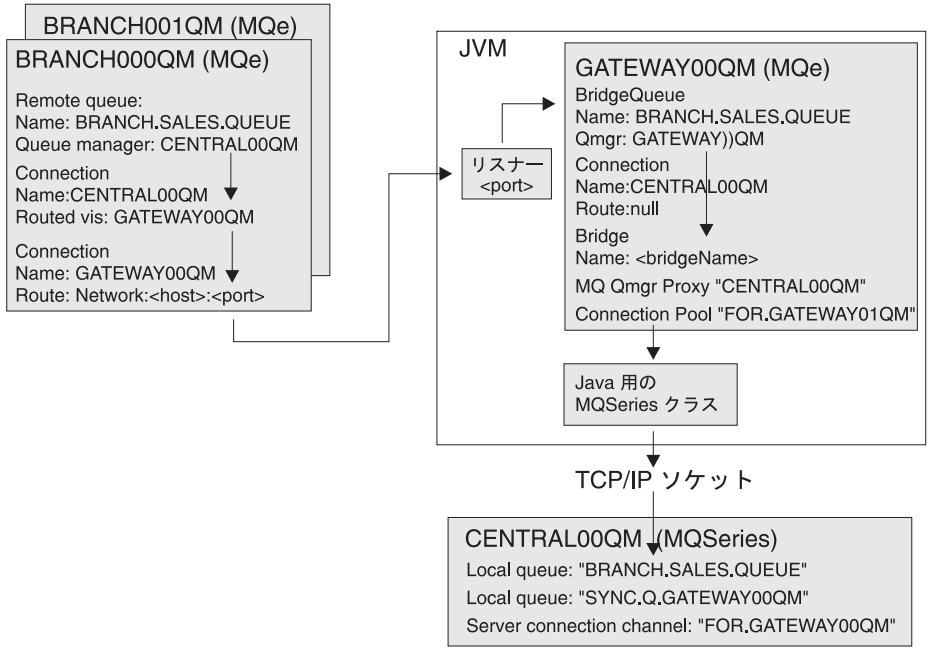

図 29. 事業所 - 本社経路指定

以下のオブジェクトは、本社 - 事業所経路指定を提供するためにスクリプトによって作成されます。

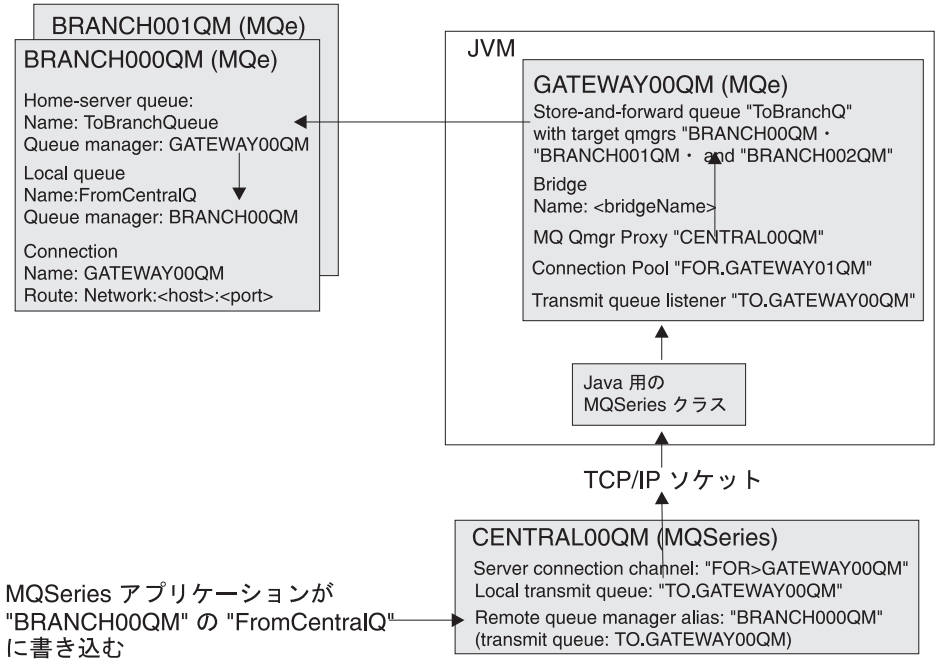

図 30. 本社 - 事業所経路指定

スクリプト・ファイルの使用法

以下の手順に従って必要なオブジェクトを作成し、提供されたファイルを使用してサンプル・シナリオを実行します。

Edit the JavaEnv.bat .

ユーザーの必須作業環境を設定するために、JavaEnv.bat ファイル編集済みであることを確認します。

コマンド行セッションを作成します。

コマンド行セッションを作成し、JavaEnv.bat を起動して、各設定値が現行環境で使用できるようにします。

必要なハードウェアの収集

ネットワーク・トポロジーをインストールするすべてのハードウェアを探し出します。

ユーザーが使用できるマシンの名前を収集し、それらを書き留めます。マシンが 1 台しかない場合でも、各キュー・マネージャーに同じホスト名を指定できるため、引き続きスクリプトを使用してサンプル・ネットワーク・トポロジーを展開することができます。

MQSeries キュー・マネージャーの作成

デフォルトで、スクリプトは、これが CENTRAL00QM と呼ばれ、クライアント・チャネル接続のためにポート 414 を listen していると見なします。

MQSeries キュー・マネージャーの記述

CentralQMDetails.bat ファイルの詳細が、ユーザーが作成した MQSeries キュー・マネージャーの詳細と一致するように、このファイルを編集したり見直したりします。スクリプト・ファイルでは、MQSeries キュー・マネージャーが常駐しているマシンの名前を除くすべての値に、デフォルトがあります。

ゲートウェイ・キュー・マネージャーの記述

ゲートウェイ キュー・マネージャーの詳細が決定されて、他の .bat ファイルで使用できるようにするために、 GatewayQMDetails.bat ファイルを編集したり見直したりします。

スクリプトによって作成されたゲートウェイ・キュー・マネージャーのデフォルト名は GATEWAY00QM です。ユーザーは、マシン名と、このマシンが listen するポートの番号を設定する必要があります。このポートは使用可能になっていなければなりません。

ヒント: Windows マシンでは、コマンド **netstat -a** を使用して、現在使用中のポートのリストを取得します。

central.tst ファイルの検討

central.tst ファイルを読み取り、ユーザーが望んでいないような MQSeries オブジェクトが、このファイルによって MQSeries キュー・マネージャー上に作成されていないことを確認します。

すべてのマシンへのすべてのスクリプトの配布

MQSeries Everyplace キュー・マネージャーが実行されるすべてのマシンにすべてのスクリプトをコピーします。

このステップは、ユーザーが使用するホスト名、ポート番号、およびキュー・マネージャー名をネットワーク内のすべてのマシンに知らせます。これらのファイルのいずれかが変更された場合は、すべての MQSeries Everyplace キュー・マネージャーを削除し、命令のこのポイントから再始動してください。

新規 MQSeries キュー・マネージャーでの central.tst スクリプトの実行

central.tst スクリプトは、 MQSeries と一緒に提供される **runmqsc** サンプル・プログラムで使用するフォーマットになっています。

central.tst ファイルを **runmqsc** にパイピングして MQSeries キュー・マネージャーを構成します。たとえば、次のとおりです。

```
runmqsc CENTRAL00QM < Central.tst
```

MQSeries Explorer を使用して、作成された MQSeries オブジェクトを表示します。

マイルストーン: ユーザーの MQSeries システムがセットアップされました。

CreateGatewayQM スクリプトの実行

CreateGatewayQM スクリプトは、 CentralQMDetails および GatewayQMDetails スクリプトの詳細を使用して ゲートウェイ・キュー・マネージャーを作成します。

スクリプトにはパラメーターは不要です。

テスト・メッセージの検査

キュー・マネージャーを作成するスクリプトは、テスト・メッセージを MQSeries システムに送信します。

MQSeries Explorer ツールを使用して宛先キュー (デフォルトでは、`BRANCH.SALES.QUEUE`) を調べ、テスト・メッセージが到着しているか確認します。テスト・メッセージの本体にはストリング `ABCD` が含まれています。

マイルストーン: これで、MQSeries Everyplace ゲートウェイ・キュー・マネージャーがセットアップされました。

ゲートウェイ・キュー・マネージャーの実行継続

CreateGatewayQM スクリプトを実行しているときに、サンプル・サーバー・プログラムを起動して ゲートウェイ・キュー・マネージャーを開始し、実行を続けます。次のウィンドウが表示されます。

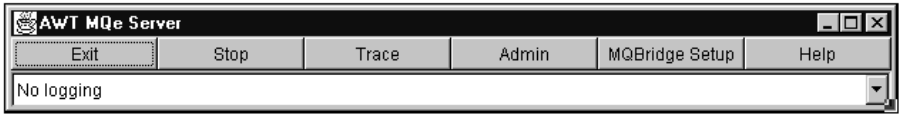

図 31. ゲートウェイ・キュー・マネージャー・ウィンドウ

このウィンドウは閉じないでください。

このウィンドウは常時アクティブになっており、またこのウィンドウに示されている MQSeries Everyplace ゲートウェイ・キュー・マネージャーもアクティブになっています。このウィンドウを閉じると MQSeries Everyplace ゲートウェイ・キュー・マネージャーが閉じられて、ブランチ・キュー・マネージャーから MQSeries キュー・マネージャーへのパスが切断されます。

ブランチ・キュー・マネージャーの作成

ブランチ・キュー・マネージャーを別のマシンで実行する必要がある場合は、ローカル環境をセットアップするよう JavaEnv.bat ファイルを編集しなければならないことがあります。

コマンド行セッションを作成し、従来どおり JavaEnv.bat を呼び出して作業用環境をセットアップします。

CreateBranchQM スクリプトを使用してブランチ・キュー・マネージャーを作成します。コマンドの構文は以下のとおりです。

```
CreateBranchQM.bat branchNumber portListeningOn
```

Where:

branchNumber
3 桁の番号で、先行ゼロが埋め込まれます。どのブランチのキュー・マネージャーを作成するかを示します。たとえば、000、001、002...。

portListeningOn
管理要求に応じるために、デバイス・ブランチ・キュー・マネージャーが listen するポート。たとえば、8082、8083...。

注: このポートは、使用中のものであってはなりません。
ヒント: Windows マシンでは、 **netstat -a** コマンドを使用して使用中のポートのリストを表示します。

スクリプト実行時には、テスト・メッセージが MQSeries システムに送信されます。 MQSeries Explorer を使用してテスト・メッセージが正常に到着しているか確認します。テスト・メッセージの本体にはストリング `ABCD` が含まれています。

スクリプトの実行が終わると、サンプル・プログラムを使用して MQSeries Everyplace キュー・マネージャーが開始されます。次のウィンドウが表示されます。

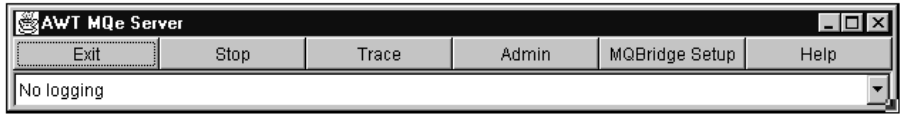

図 32. ブランチ・キュー・マネージャー・ウィンドウ

ゲートウェイ・キュー・マネージャーの場合と同様、***キュー・マネージャーを閉じるまで、このウィンドウを閉じないでください。***

ブランチ・キュー・マネージャーの展開
ブランチ・キュー・マネージャーは、その作成時に指定されたポートに、チャネル・マネージャーおよびリスナーとともにセットアップされ、1 次ネットワーク接続は HttpTcpipAdapter になります。その結果、 MQe_Explorer を使用してキュー・マネージャーを表示することができます。(172ページの『MQe_Explorer を使用した構成の表示』を参照してください)。

マイルストーン: これで、ブランチ・キュー・マネージャーがセットアップされました。

注: MQSeries Everyplace キュー・マネージャーの名前は固有でなければなりません。同じ名前を持つ 2 つのキュー・マネージャーを作成しないでください。

MQe_Explorer.exe プログラムを開始します。ブランチ・キュー・マネージャーのうちの 1 つ、たとえば BRANCH002QM を停止します。BRANCH002QM.ini ファイルを開き、そこからナビゲートします。

MQe_Explorer を使用した構成の表示

MQe_Explorer を使用して構成を表示するには、次のようにします。

1. MQe_Explorer.exe プログラムを開始します。
2. ブランチ・キュー・マネージャーのうちの 1 つ、たとえば BRANCH002QM を停止します。
3. BRANCH002QM.ini ファイルを開き、そこからナビゲートします。

第7章 MQSeries-ブリッジ

MQSeries-ブリッジは、MQSeries Everyplace ネットワークが MQSeries ネットワークとインテリジェントなメッセージ交換をできるようにするためのソフトウェアの一部です。それぞれのシステムには異なる要件を満たす目的があるため、この 2 つのシステムがメッセージを渡す方法にも相違があります。ブリッジは、これらの相違を解決し、さまざまなシステムの間でメッセージが流れることを可能にするものです。

インストール

ブリッジ・コードは、 MQeMQBridge.jar ファイルにパッケージされています。また、クラス・ファイルは、 com¥ibm¥mqe¥mqbridge ディレクトリーにもあります。クラスパスは、MQSeries Everyplace サーバー の始動時に、ブリッジ・クラスがアクセス可能になるようにセットアップしなければなりません。 MQSeries-ブリッジ・コードは、デバイス・プラットフォームではなく MQSeries Everyplace ゲートウェイ・プラットフォームでのみ稼働します。

MQSeriesJava クラス

MQSeries-ブリッジを使用するには、 MQSeries Java クラス (バージョン 5.1 またはそれ以上) を MQSeries Everyplace システムにインストールする必要があります。MQSeries Java クラスは、 supportpac MA88 として、 Web サイトから無料でダウンロードすることができます。ダウンロードのための Web アドレスは、 `http://www.ibm.com/software/mqseries/txppacs/ma88.html` です。 (MQSeries Java クラス for NT は、 MQSeries バージョン 5.1 for NT に付属して提供されています。)

MQSeries-ブリッジの構成

MQSeries-ブリッジを構成するには、 MQSeries キュー・マネージャーおよび MQSeries Everyplace キュー・マネージャーでいくつかの操作を行う必要があります。理論的には、ブリッジはメッセージの送信方向ごとに 2 つの部分に分かれています (MQSeries Everyplace から MQSeries へ、および MQSeries から MQSeries Everyplace へ)。

175ページの図33 が示すように、ブリッジ・オブジェクトは階層として定義されます。

さまざまなオブジェクトの関係は、以下のルールによって制御されます。

- 1 つの MQSeries Everyplace ブリッジ・オブジェクトは単一の MQSeries Everyplace キュー・マネージャーに関連付けられます。
- 単一の MQSeries Everyplace ブリッジ・オブジェクトには、複数のブリッジ・オブジェクトを関連付けることができます。それぞれ異なる経路指定を使って複数の MQSeries-ブリッジ インスタンスを構成することもできます。

- 各ブリッジには、複数の MQSeries キュー・マネージャーのプロキシーを定義することができます。
- それぞれの MQSeries キュー・マネージャーのプロキシー定義には、 MQSeries Everyplace と通信可能なクライアント接続を複数設定することができます。
- それぞれのクライアント接続は、単一の MQSeries キュー・マネージャーに接続します。ただし、各接続ごとに MQSeries キュー・マネージャーとの間で別々のサーバー接続 (つまり、異なるセキュリティー、送信出口と受信出口、ポートその他のパラメーターの組み合わせ) を使用することもできます。
- 1 つの MQSeries-ブリッジ・クライアント接続につき、そのブリッジ・サービスを使って MQSeries キュー・マネージャーへ接続する複数の伝送キュー・リスナーを設定できます。
- 1 つのリスナーは、接続を確立するためにクライアント接続を 1 つだけ使用します。
- 各リスナーは MQSeries システム上の単一の伝送キューに接続します。
- それぞれのリスナーは、(ブリッジに関連した MQSeries Everyplace キュー・マネージャーを使って) 単一の MQSeries 伝送キューから MQSeries Everyplace ネットワークの任意の位置にメッセージを移動します。したがって、各 MQSeries-ブリッジでは、1 つの MQSeries Everyplace キュー・マネージャーを介して複数の MQSeries メッセージ・ソースを MQSeries Everyplace ネットワークに送ることができます。
- MQSeries Everyplace メッセージを MQSeries ネットワークに移動するとき、MQSeries Everyplace キュー・マネージャーは複数のアダプター・オブジェクトを作成します。それぞれのアダプター・オブジェクトは、(構成されていれば) 任意の MQSeries キュー・マネージャーに接続して、任意のキューにメッセージを送信することができます。このように 1 つの MQSeries-ブリッジを使って、単一の MQSeries Everyplace キュー・マネージャーを経由して、任意の MQSeries キュー・マネージャーへ MQSeries Everyplace メッセージを送ることができます。

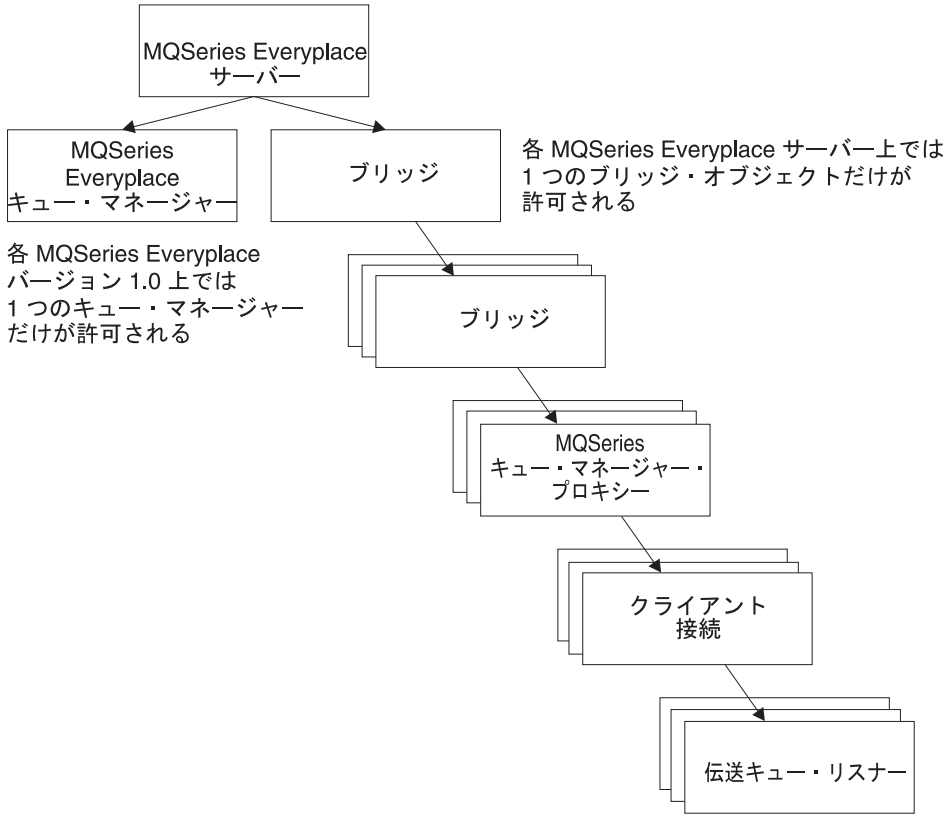

図 33. ブリッジのオブジェクト階層

基本インストールの構成

MQSeries-ブリッジの基本的なインストールを構成するには、以下のステップを完了する必要があります。

1. MQSeries システムがインストールされていることを確認し、ローカルの経路指定規則とシステム構成方法を理解していることを確かめます。
2. MQSeries Everyplace をシステムにインストールします (MQSeries システムと同じシステムにインストールすることができます)。
3. MQSeries Java クラスをまだインストールしていない場合は、Web からダウンロードしてインストールします。 (173ページの『MQSeriesJava クラス』を参照してください。)
4. MQSeries への接続を試行する前に、MQSeries Everyplace システムをセットアップして正常に作動することを確かめます。
5. MQe_java¥Classes¥JavaEnv.bat ファイルを更新して、MQSeries Java クラスに含まれる Java クラス、および JRE (Java Runtime Environment) へのクラスパスを指示する

ようにします。クラス com.ibm.mqbind.jar および com.ibm.mq.jar を必ずクラスパスに含め、 java¥lib および ¥bin ディレクトリーがパスに含まれるようにしてください。

6. インプリメントする経路指定について計画します。どの MQSeries キュー・マネージャーがどの MQSeries Everyplace キュー・マネージャーと通信するかを決める必要があります。
7. MQSeries Everyplace オブジェクトと MQSeries オブジェクトの命名規則を決定して、将来の利用のために書き留めておきます。
8. MQSeries Everyplace システムを変更して、選択した MQSeries Everyplace サーバー上の MQSeries-ブリッジを定義します。 examples.mqbridge.awt.AwtMQBridgeServer を使用してブリッジを定義する方法については、 187ページの『管理 GUI アプリケーションの例』を参照してください。
9. 以下のように、選択された MQSeries キュー・マネージャーを MQSeries Everyplace サーバー上のブリッジに接続します。
   - MQSeries キュー・マネージャーでは、以下の事柄を行います。

     1 つまたは複数の Java サーバー接続チャネルを定義して、このキュー・マネージャーと通信する Java MQSeries クラスを MQSeries Everyplace が使用できるようにします。これには、以下のステップが含まれます。

     a. (1 つまたは複数の) サーバー接続チャネルの定義。
     b. MQSeries Everyplace が使用する同期キューを定義して、MQSeries Everyplace が確実に MQSeries システムに送信できるようにする。サーバーごとのこれらの接続チャネルのうち、MQSeries Everyplace システムの使用するチャネルが 1 つ必要です。

   - MQSeries Everyplace サーバーでは、以下の事柄を行います。

     a. MQSeries キュー・マネージャーに関する情報を収める MQSeries キュー・マネージャー・プロキシー・オブジェクトを 1 つ定義します。これには、以下のステップが含まれます。
        1) MQSeries キュー・マネージャーのホスト名を収集します。
        2) そのホスト名を MQSeries キュー・マネージャー・プロキシー・オブジェクトの中に含めます。
     b. Java MQSeries クラスを使用して、 MQSeries システムのサーバー接続チャネルに接続する方法に関する情報を収めたクライアント接続オブジェクトを定義します。これには、以下のステップが含まれます。
        1) ポート番号その他のサーバー接続チャネル・パラメーターを取得します。
        2) それらが MQSeries キュー・マネージャーの定義と適合するように、これらの値を使用してクライアント接続オブジェクトを定義します。
10. MQSeries Everyplace および MQSeries の構成を変更して、メッセージが MQSeries から MQSeries Everyplace へ送られるようにします。

a. MQSeries から MQSeries Everyplace ネットワークへの経路の数を決定します。必要な経路の数は、それぞれの経路で予定しているメッセージ・トラフィック (負荷) の量によって決まります。メッセージ負荷が大きい場合には、トラフィックを多数の経路に分割することができます。

b. 以下のようにして経路を定義します。

1) MQSeries システムに定義された伝送キューが、それぞれの経路ごとに 1 つ必要です。これらの伝送キューに対して、チャネルを定義しないでください。

2) MQSeries Everyplace システムに作成された突き合わせ伝送キューが、それぞれの経路ごとに 1 つ必要です。

3) 一連のリモート・キュー定義 (リモート・キュー・マネージャー別名、キュー別名など) を定義して、 10b1 のステップで定義した MQSeries Everyplace 向けのさまざまな伝送キューに MQSeries メッセージが経路指定されるようにします。

11. MQSeries Everyplace の構成を変更して、メッセージが MQSeries Everyplace から MQSeries へ送られるようにします。

a. MQSeries Everyplace ネットワークからメッセージを送る必要のある送信先 MQSeries ネットワーク上の、すべてのキュー・マネージャーに関する詳細情報を公表します。各 MQSeries キュー・マネージャーごとに、MQSeries Everyplace サーバー上で接続を定義する必要があります。 このキュー・マネージャーとの会話に通常の MQSeries Everyplace 通信チャネルを使用しないことを示すために、キュー・マネージャー名を除くすべてのフィールドをヌルにする必要があります。

b. MQSeries Everyplace ネットワークからメッセージを送る必要のある送信先 MQSeries ネットワーク上の、すべてのキューに関する詳細情報を公表します。各 MQSeries キューごとに、MQSeries Everyplace サーバー上で MQSeries-ブリッジ キューを定義する必要があります。 (これは `DEFINE QREMOTE` に対応する MQSeries 内の MQSeries Everyplace です)。

- キュー名は、この MQSeries-ブリッジ キューに到着するすべてのメッセージをブリッジが送る送信先 MQSeries キューの名前です。
- キュー・マネージャー名は、キューが格納される MQSeries キュー・マネージャーの名前です。
- ブリッジ名は、MQSeries ネットワークへのメッセージ送信に使用されるブリッジを示します。
- MQSeries キュー・マネージャー・プロキシー名は、MQSeries Everyplace 構成における、 MQSeries キュー・マネージャーに接続可能な MQSeries キュー・マネージャー・プロキシー・オブジェクトの名前です。
- この MQSeries キュー・マネージャーには、経路を定義する必要があります。こうすると、メッセージがキュー・マネージャー名上のキュー名に通知されて、最終的な宛先に送られるようになります。

12. MQSeries システムおよび MQSeries Everyplace システムを開始して、メッセージが流れるようにします。 MQSeries システム・クライアント・チャネル・リスナーが開始されなければなりません。 MQSeries Everyplace では、定義済みのすべてのオブジェクトが開始されなければなりません。これらのオブジェクトは、以下のいずれかの方法で開始することができます。
    - 187ページの『管理 GUI アプリケーションの例』で説明されている管理 GUI を明示的に使用する
    - 始動状態 (実行または停止) を示すためにルール・クラスを構成し (213ページの『MQSeries-ブリッジのルール』で説明されている)、 MQSeries Everyplace サーバーを再始動する
    - 上記の 2 つのメソッドを両方行う

    オブジェクトを手動で開始する最も単純な方法は、 **start** コマンドを関係のあるブリッジ・オブジェクトに送信することです。このコマンドは、ブリッジのすべての子、および子の子も開始されなければならないことを示します。
    - メッセージが MQSeries Everyplace から MQSeries に渡るようにするには、MQSeries Everyplace 内のクライアント接続オブジェクトを開始します。
    - メッセージが MQSeries から MQSeries Everyplace に渡るようにするには、クライアント接続オブジェクト、および関連する伝送キュー・リスナーの両方を開始します。
13. 変換機能クラスを作成し、MQSeries Everyplace がそれらを使用するように構成を変更します。変換機能クラスは、MQSeries メッセージ・フォーマットを MQSeries Everyplace メッセージ・フォーマットに、またはその逆に変換します。これらのフォーマット変換機能は Java で作成して、 MQSeries-ブリッジ 構成内の数か所に配置する必要があります。
    a. Java の変換機能クラスを作成する方法
        - ブリッジを介して渡す必要のある MQSeries メッセージのメッセージ・フォーマットを判別します。
        - MQSeries メッセージ・フォーマットを MQSeries Everyplace メッセージに変換する、一連の変換機能クラスを作成します。208ページの『変換機能』を参照してください。

    b. デフォルトの変換機能クラスを置換することができます。そうするには、管理用 GUI を使用して、ブリッジ・オブジェクト構成内のデフォルト変換機能クラス・パラメーターを**更新**します。

    c. それぞれの MQSeries-ブリッジ・キュー定義ごとに、デフォルト以外の変換機能を 1 つずつ指定することができます。そうするには、管理用 GUI を使用して、構成内のそれぞれの MQSeries-ブリッジ・キューの***変換機能***フィールドを**更新**します。

d. それぞれの MQSeries 伝送キュー・リスナーごとに、デフォルト以外の変換機能を 1 つずつ指定することもできます。そうするには、管理用 GUI を使用して、構成内のそれぞれのリスナーの*変換機能*フィールドを**更新**します。
e. ブリッジおよびすべてのリスナーを再始動します。

サンプル構成ツール

MQSeries Everyplace システムと MQSeries-ブリッジは、複合的な環境です。初期状態の構成を作成するのに役立つサンプル構成ツールが、MQSeries-ブリッジに付属しています。このツールのソース・コードが提供されています。必要に応じてこれをサブクラス化、修正、または動作変更することができます。

ここでは、このサンプル・ツールの機能と使用方法を説明します。

制限

このサンプル構成ツールは、多数の MQSeries Everyplace キュー・マネージャー接続が定義されているサーバーでは使用できません。たとえば、それぞれ別個のキュー・マネージャーに関連付けられた多数の携帯電話が存在して、サーバーにそれぞれの接続が定義されている場合には、正常には動作しません。それは、ツールが時折、接続のリストを照会するためであり、そのような状況では、メモリー不足のためにツールが停止して、 JVM が停止します。他の MQSeries Everyplace キュー・マネージャーへの接続が多数存在するサーバーを管理しようとしている場合には、代わりに `examples.mqbridge.administration.console.AdminGateway` アプリケーションを使用することをお勧めします。

ブリッジの構成に必要なステップ

MQSeries-ブリッジの基本的なインストールを構成するには、 175ページの『基本インストールの構成』のステップを完了する必要があります。サンプル・ツールは、このリストのステップ 8 からステップ 12 までを簡素化するために提供されています。

構成の例

このセクションでは、4 つのシステムにおける構成の例を示します。

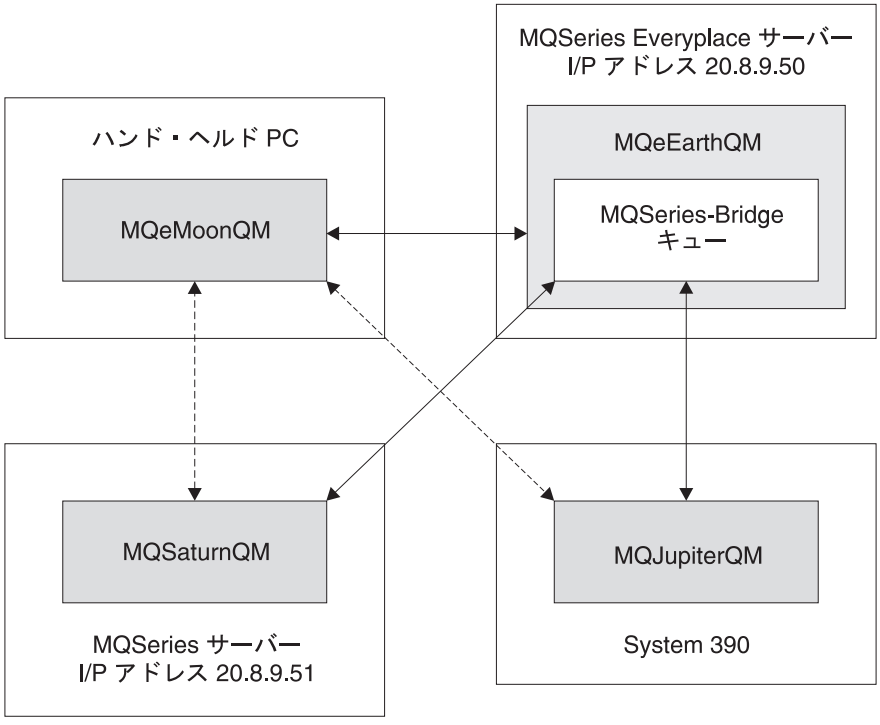

図 34. 構成の例

4 つのシステムは次のとおりです。

MQeMoonQM
携帯用 PC をサイトとする MQSeries Everyplace クライアント・キュー・マネージャーです。ユーザーは携帯用 PC を定期的にネットワークに接続して、MQeEarthQM MQSeries Everyplace ゲートウェイと連動させます。

MQeEarthQM
I/P アドレスが 20.8.9.50 である Windows 2000 マシン上で実行されます。これは MQSeries Everyplace ゲートウェイ (サーバー) キュー・マネージャーです。

MQSaturnQM
Windows NT プラットフォームにインストールされた MQSeries キュー・マネージャーです。 I/P アドレスは 20.8.9.51 です。

MQJupiterQM
System/390 プラットフォームにインストールされた MQSeries キュー・マネージャーです。

要件

この例の要件は、すべてのマシンが他の任意のマシン上のキューにメッセージを通知できることです。

断続的に接続される MQeMoonQM マシンを除いて、すべてのマシンがネットワークに継続的に接続されていることを想定します。

初期セットアップ

この例の場合、メッセージを収めることのできるローカル・キューが、すべてのキュー・マネージャーに存在することを想定しています。これらのキューは次のとおりです。

- MQeMoonQ (MQeMoonQM 上)
- MQeEarthQ (MQeEarthQM 上)
- MQSaturnQ (MQSaturnQM 上)
- MQJupiterQ (MQJupiterQM 上)

MQeMoonQM が MQeEarthQM キュー・マネージャーとの間でメッセージを送受信できるようにする

MQeMoonQM 上で

1. 次のパラメーターで**接続**を定義します。

   宛先キュー・マネージャー名 : MQeEarthQM
   アダプター : Network:20.8.9.50

   これで、MQeMoonQM がネットワークに接続されている間、アプリケーションは MQeEarthQM キュー・マネージャー上に定義された任意のキューを直接使用できます。要件は、MQeMoonQM 上のアプリケーションが非同期的に MQeEarthQ にメッセージを送信できなければならないというものです。これにより、MQeEarthQ キューに非同期リンケージをセットアップするためのリモート・キュー定義が必要となります。

2. 次のパラメーターで**リモート・キュー**を定義します。

   キュー名 : MQeEarthQ
   キュー・マネージャー名 : MQeEarthQM
   アクセス・モード : Asynchronous

   これで、MQeMoonQM 上のアプリケーションは MQeMoonQ (ローカル・キュー) に同期的にアクセスし、 MQeEarthQ に非同期的にアクセスすることができます。

MQeEarthQM が MQeMoonQM キュー・マネージャーにメッセージを送信できるようにする

MQeMoonQM はまれにしかネットワークに接続されないため、 MQeEarthQM 上でストア・アンド・フォワード (蓄積交換) キューを使用して、 MQeMoonQM あてのメッセージを収集します。

MQeEarthQM 上で

1. 次のパラメーターで**ストア・アンド・フォワード (蓄積交換)** を定義します。

   キュー名 : TO.HANDHELDS
   キュー・マネージャー名 : MQeEarthQM

2. 次のパラメーターを使用して、**ストア・アンド・フォワード (蓄積交換) キュー**に**キュー・マネージャー**を追加します。

   キュー名: TO.HANDHELDS
   キュー・マネージャー : MQeMoonQM

(同様の名前の) ホスト・サーバー・キューが、 MQeMoonQM キュー・マネージャー上に必要です。このキューはストア・アンド・フォワード (蓄積交換) キューからメッセージを取り出して、それらを MQeMoonQM キュー・マネージャー上のキューに書き込みます。

MQeMoonQM 上で

1. 次のパラメーターで**ホーム・サーバー・キュー**を定義します。

   キュー名 : TO.HANDHELDS
   キュー・マネージャー名 : MQeEarthQM

MQeMoonQM あてのメッセージが MQeEarthQM に到着すると、すべてストア・アンド・フォワード (蓄積交換) キュー TO.HANDHELDS に一時的に保管されます。 MQeMoonQM が次にネットワークに接続した時点で、ホーム・サーバー・キュー TO.HANDHELDS はストア・アンド・フォワード (蓄積交換) キューにあるメッセージを取り出して MQeMoonQM キュー・マネージャーに送ります。その後、メッセージはローカル・キューに保管されます。

これで、MQeEarthQM 上のアプリケーションはメッセージを MQeMoonQ に非同期的に送ることができます。

MQeEarthQM が MQSaturnQ にメッセージを送信できるようにする

MQeEarthQM 上で

1. 次のパラメーターで**ブリッジ**を定義します。

   ブリッジ名 : MQeEarthQMBridge

2. 次のパラメーターで**MQSeries キュー・マネージャー・プロキシー**を定義します。

   ブリッジ名 : MQeEarthQMBridge
   MQ QMgr プロキシー名 : MQSaturnQM
   ホスト名 : 20.8.9.51

3. 次のパラメーターで**クライアント接続**を定義します。

   ブリッジ名 : MQeEarthQMBridge
   MQ QMgr プロキシー名 : MQSaturnQM
   ClientConnectionName : MQeEarth.CHANNEL
   SyncQName : MQeEarth.SYNC.QUEUE

4. 次のパラメーターで**接続**を定義します。

   ConnectionName : MQeSaturnQM
   チャネル : ヌル
   アダプター : ヌル

5. 次のパラメーターで**MQSeries-ブリッジ・キュー**を定義します。

   キュー名 : MQSaturnQ
   MQ キュー・マネージャー名 : MQSaturnQM
   ブリッジ名 : MQeEarthQMBridge
   MQ QMgr プロキシー名 : MQSaturnQM
   ClientConnectionName : MQeEarth.CHANNEL

MQSaturnQM 上で

1. 次のパラメーターで**サーバー接続チャネル**を定義します。

   名前 : MQeEarth.CHANNEL

2. 次のパラメーターで**ローカル同期キュー**を定義します。

   名前 : MQeEarth.SYNC.QUEUE

   確実な送信のためには、同期キューが必要です。

これで、MQeEarthQM 上のアプリケーションは MQSaturnQM 上の MQSaturnQ にメッセージを送信できます。

MQeEarthQM が MQJupiterQ にメッセージを送信できるようにする

MQeEarthQM 上で

1. 次のパラメーターで**接続**を定義します。

ConnectionName : MQeJupiterQM
チャネル : ヌル
アダプター : ヌル

2. 次のパラメーターで**MQSeries-ブリッジ・キュー**を定義します。

キュー名 : MQJupiterQ
MQ キュー・マネージャー名 : MQJupiterQM
ブリッジ名 : MQeEarthQMBridge
MQ QMgr プロキシー名 : MQSaturnQM
ClientConnectionName : MQeEarth.CHANNEL

MQSaturnQM 上で

1. 次のパラメーターで**リモート・キュー定義**を定義します。

キュー名 : MQJupiterQ
伝送キュー : MQJupiterQM.XMITQ

MQSaturnQM および MQJupiterQM の両方で

1. MQSaturnQM 上の MQJupiterQM.XMITQ から MQJupiterQM にメッセージを移動するための**チャネル**を定義します。

これで、MQeEarthQM 上のアプリケーションは MQSaturnQM を介して MQJupiterQM 上の MQJupiterQ にメッセージを送信できます。

MQeMoonQM が MQJupiterQ および MQSaturnQ にメッセージを送信できるようにする

MQeMoonQM 上で

1. 次のパラメーターで**接続**を定義します。

宛先キュー・マネージャー名 : MQSaturnQM
アダプター : MQeEarthQM

MQSaturnQM キュー・マネージャー向けのすべてのメッセージは MQeEarthQM キュー・マネージャーを介して送られることが、この接続によって示されています。

2. 次のパラメーターで**リモート・キュー定義**を定義します。

キュー名 : MQSaturnQ
キュー・マネージャー名 : MQSaturnQM
アクセス・モード : Asynchronous

3. 次のパラメーターで**接続**を定義します。

宛先キュー・マネージャー名 : MQJupiterQM
アダプター : MQeEarthQM

4. 次のパラメーターで**リモート・キュー定義**を定義します。

   キュー名 : MQJupiterQ
   キュー・マネージャー名 : MQJupiterQM
   アクセス・モード : Asynchronous

これで、MQeMoonQM に接続したアプリケーションは、携帯用 PC がネットワークから切断されていても、 MQeMoonQ、MQeEarthQ、MQSaturnQ、および MQJupiterQ のすべてに向けてメッセージを発信できるようになりました。

MQSaturnQM が MQeEarthQ にメッセージを送信できるようにする

MQSaturnQM 上で

1. 次のパラメーターで**ローカル・キュー**を定義します。

   キュー名 : MQeEarth.XMITQ
   キュー・タイプ : 伝送キュー

2. 次のパラメーターで**キュー・マネージャー別名** (リモート・キュー定義) を定義します。

   キュー名 : MQeEarthQM
   リモート・キュー・マネージャー名 : MQeEarthQM
   伝送キュー : MQeEarth.XMITQ

MQeEarthQM 上で

1. 次のパラメーターで**伝送キュー・リスナー**を定義します。

   ブリッジ名 : MQeEarthQMBridge
   MQ QMgr プロキシー名 : MQSaturnQM
   ClientConnectionName : MQeEarth.CHANNEL
   リスナー名 : MQeEarth.XMITQ

これで、MQSaturnQM 上のアプリケーションは MQeEarthQM キュー・マネージャー別名を使って MQeEarthQ へメッセージを送信できます。各メッセージは MQeEarth.XMITQ に経路指定され、そこで MQSeries Everyplace 伝送キュー・リスナー MQeEarth.XMITQ がメッセージを収集した後、 MQSeries Everyplace ネットワークに向けて移動します。

MQSaturnQM が MQeMoonQ にメッセージを送信できるようにする

MQSaturnQM 上で

1. 次のパラメーターで**キュー・マネージャー別名** (リモート・キュー定義) を定義します。

キュー名 : MQeMoonQM
リモート・キュー・マネージャー名 : MQeMoonQM
伝送キュー : MQeEarth.XMITQ

これで、MQSaturnQM 上のアプリケーションは MQeMoonQM キュー・マネージャー別名を使って MQeMoonQ へメッセージを送信できます。各メッセージは MQeEarth.XMITQ に経路指定され、そこで MQSeries Everyplace 伝送キュー・リスナー MQeEarth.XMITQ がメッセージを収集した後、MQSeries Everyplace ネットワークに向けて通知します。

ストア・アンド・フォワード (蓄積交換) キュー TO.HANDHELDS がメッセージを収集し、 MQeMoonQM が次にネットワークに接続した時点で、ホーム・サーバー・キューがストア・アンド・フォワード (蓄積交換) キューからメッセージを取得して、それらを MQeMoonQ に送ります。

MQJupiterQM が MQeMoonQ にメッセージを送信できるようにする

MQJupiterQM 上で

MQeEarthQM および MQeMoonQM の**リモート・キュー・マネージャー別名**を設定して、メッセージが通常の MQSeries 経路指定手法によって MQSaturnQM に経路指定されるようにします。

これで、すべてのキュー・マネージャーに接続するすべてのアプリケーションは、MQeMoonQ、MQeEarthQ、MQSaturnQ、または MQJupiterQ のいずれに対してもメッセージを通知できるようになりました。

追加のブリッジ構成

通常、基本 MQSeries Java クラスのトレースは必要ではないため、デフォルトでは使用不可になっています。ただし、MQSeries トレースの初期設定はアクティブ・トレース・ハンドラー・クラスで行う必要があり、初期設定方法の例が MQSeries Everyplace クラスに付属して提供されています。ブリッジ・トレース・クラスの例は examples.mqbridge.awt.AwtBridgeTrace です。このクラスは、ブリッジ管理用 GUI によって自動的にインスタンス化されます (187ページの『管理 GUI アプリケーションの例』を参照してください)。 MQSeries-ブリッジ・トレース・メッセージは、いくつかの言語で examples.mqbridge.trace に収められています。

さらに、com.ibm.mq.MQException.log (デフォルトでは System.err) に定義された OutputStreamWriter に MQExceptions が記録されます。基本 MQSeries トレースの初期設定および構成について、詳しくは「*MQSeries Java の使用*」の資料を参照してください。

MQSeries-ブリッジの管理

このセクションには、MQSeries-ブリッジの管理に関連した作業についての情報が含まれています。

管理 GUI アプリケーションの例

MQSeries-ブリッジには管理 GUI の例が提供されています。これは、156ページの『管理コンソールの例』で説明されている examples.administration.console.Admin のサブクラスの例です。

サブクラスは examples.mqbridge.administration.console.AdminGateway と呼ばれます。

MQSeries-ブリッジ機能はクライアント・キュー・マネージャーでは実行することができません。そのため、このクラスをクライアント・キュー・マネージャーと共に使用しても、そのクライアント・キュー・マネージャー上でブリッジ・オブジェクトを管理することはできません。しかし、リモート MQSeries-ブリッジが使用可能なサーバー・キュー・マネージャーの管理は行えます。

ローカル・キュー・マネージャーに接続された MQSeries-ブリッジを管理するには、サーバー・プログラムの例 `<java> examples.mqbridge.awt.AwtMQBridgeServer <server_ini_file>` を使用して、MQSeries Everyplace サーバーを開始します。

図 35. MQSeries-ブリッジ管理 GUI サーバー・ウィンドウ

サーバー・ウィンドウから、次のどちらかのオプションを使用できます。

- **「管理 (Admin)」**ボタンをクリックして、examples.mqbridge.administration.console.AdminGateway クラスを使用し、ローカル・サーバー・キュー・マネージャーを管理する。これがブリッジ・オブジェクトです。
- **「MQBridge のセットアップ (MQBridge Setup)」**ボタンをクリックして、179ページの『サンプル構成ツール』で説明されているように examples.mqbridge.setup.MQBridgeWizard クラスの例を呼び出す。

どちらの例も、ブリッジ固有の管理メッセージのサブクラス (MQeMQBridgesAdminMsg、 MQeMQBridgeAdminMsg、 MQeMQQMgrProxyAdminMsg、 MQeClientConnectionAdminMsg、 MQeListenerAdminMsg、および MQeMQBridgeQueueAdminMsg) を使って、 MQSeries-ブリッジ構成オブジェクトを方針に基づいて操作する方法を示しています。

MQSeries-ブリッジ管理アクション

実行状態

管理オブジェクトには、それぞれ*実行可能状態* があります。そのオブジェクトがアクティブであれば実行中であり、そうでなければ停止になります。

管理オブジェクトが停止状態になっているときには、それを使用することはできません。しかしそのオブジェクトの構成パラメーターを照会したり更新したりすることはできます。

MQSeries-ブリッジのキューが停止状態のブリッジ管理オブジェクトを表す場合、ブリッジ、MQSeries キュー・マネージャー・プロキシー、およびクライアント接続管理オブジェクトがすべて開始されるまでは、 MQSeries Everyplace メッセージを MQSeries ネットワークに送ることはできません。

管理オブジェクトの実行状態は、 MQeMQBridgeAdminMsg、MQeMQQMgrProxyAdminMsg、 MQeClientConnectionAdminMsg または MQeListenerAdminMsg 管理メッセージ・クラスから**開始**および**停止**アクションを使って変更することができます。

この後のセクションでは、MQSeries-ブリッジ管理オブジェクトがサポートするアクションについて説明します。

開始アクション

管理者は**開始**アクションをどの管理オブジェクトにも送ることができます。

affect children ブール・フラグは、このアクションの結果に影響を与えます。 *affect children* ブール・フィールドがメッセージに含まれており、それが `true` に設定されている場合、**開始**アクションによって管理オブジェクトとその子 (および子の子) がすべて開始されます。このフラグがメッセージに含まれていないか、それが `false` に設定されている場合、開始アクションを受け取る管理オブジェクトだけがその実行状態を変更します。たとえば、ブリッジ・オブジェクトに**開始**アクションを送り、 *affect children* を `true` に指定すると、すべてのプロキシー、クライアント接続、および親元であるリスナーがすべて開始されます。 *affect children* が指定されていない場合、ブリッジだけが開始されます。管理オブジェクトの親管理オブジェクトがまだ開始されていない場合、管理オブジェクトを開始することはできません。そのため、開始イベントを管理オブジェクトに送ると、階層の上位にあるすべてのオブジェクトがまだ実行されていなければ、それらが開始されます。

停止アクション

管理オブジェクトに**停止**アクションを送ると、そのオブジェクトを停止することができます。受信側の管理オブジェクトは、必ず階層の下位にあるすべてのオブジェクトが停止していることを確かめてから、自分自身を停止させます。

問い合わせアクション

問い合わせアクションは、送信先の管理オブジェクトからの値を照会します。

管理オブジェクトが実行中状態になっている場合、問い合わせによって戻される値は現在使用されているものです。停止状態のオブジェクトから戻される値は、**更新**アクションによって値に加えられた最新の変更を反映します。このように、**開始**、**更新**、**問い合**

わせという一連のアクションでは、更新*前* に構成された値が戻されますが、**開始、更新、停止、問い合わせ**という一連のアクションでは、更新*後* に構成された値が戻されます。

可変値を更新する前に管理オブジェクトを停止すれば、混乱は少なくて済むでしょう。

更新アクション

更新アクションでは、管理オブジェクトの特性に関する 1 つまたは複数の値を更新します。**更新**アクションで設定された値は、管理オブジェクトが次に停止されるまで、現在の値として使用されません。 (188ページの『問い合わせアクション』を参照してください。)

削除アクション

削除アクションでは、管理オブジェクトに関する現在の情報と永続的な情報をすべて削除します。 *affect children* ブール・フラグは、このアクションの結果に影響を与えます。 *affect children* フラグを使用しており、それが `true` に設定されている場合、このアクションを受け取る管理オブジェクトは**停止**アクションを出します。次に、階層内で管理オブジェクトの下位にあるすべてのオブジェクトに対して**削除**アクションがとられ、こうして 1 つのアクションで階層の全部分が削除されます。フラグを使用していないか、それが `false` に設定されている場合、管理オブジェクトだけが削除されます。しかし、このアクションは、階層内の現行のオブジェクトの下位にあるすべてのオブジェクトが削除されるまで実行できません。

作成アクション

作成アクションは、管理オブジェクトを作成します。作成された管理オブジェクトの実行状態は最初は停止に設定されます。

MQSeries キュー・マネージャーのシャットダウンの際の MQSeries-ブリッジの考慮事項

MQSeries キュー・マネージャーを停止する前に、すべての MQSeries キュー・マネージャー・プロキシーのブリッジ管理オブジェクトに対して **stop** 管理メッセージを発行することをお勧めします。これにより、MQSeries Everyplace ネットワークは MQSeries キュー・マネージャーを使用しようとするのをやめ、それとともに恐らく MQSeries キュー・マネージャーのシャットダウンを妨害しなくなるでしょう。 (このことは、**stop** 管理メッセージを MQebridges オブジェクトに対して 1 回発行することによって行うこともできます。)

MQSeries キュー・マネージャーをシャットダウンする前に MQSeries キュー・マネージャー・プロキシーのブリッジ・オブジェクトを停止しない場合、 MQSeries シャットダウンと MQSeries-ブリッジの動作は、選択する MQSeries キュー・マネージャー・シャットダウンのタイプ、つまり即時シャットダウンと制御シャットダウンによって異なります。

即時シャットダウン

即時シャットダウンを使って MQSeries キュー・マネージャーを停止すると、MQSeries-ブリッジと MQSeries キュー・マネージャーとの間のすべての接続が切断されます (これは、 Java バインディングまたは Java クライアント・モードのいずれかを使って構成される接続に当てはまります)。 MQSeries システムは通常どおりシャットダウンを行います。

即時シャットダウンを行うと、すべての MQSeries-ブリッジ伝送キュー・リスナーは即時に停止されます。このとき、各リスナーには MQSeries キュー・マネージャーの停止によって、伝送がシャットダウンされることが警告されます。

アクティブになっている MQSeries-ブリッジ・キューは、以下の状態になるまで、MQSeries キュー・マネージャーへの (中断した) 接続を保存します。

- アイドル・タイムアウト期間 (クライアント接続ブリッジ・オブジェクトで指定された期間) の間アイドル状態になった後の接続タイムアウト。この時点で、中断した接続はクローズされます。
- MQSeries-ブリッジ・キューには、中断した接続を使用しようとする何らかのアクション (たとえば、MQSeries に対してメッセージを送る) をとるように通知が出されます。 **putMessage** 操作は失敗し、中断した接続はクローズされます。

MQSeries-ブリッジ・キューに接続がない場合、そのキューに対して次の操作を行うと、新しい接続が獲得されます。 MQSeries キュー・マネージャーが使用可能になっていない場合、そのキューに対する操作はすぐに失敗します。 MQSeries キュー・マネージャーをシャットダウン後に再始動しており、ブリッジ・キューに対してキュー操作 (**putMessage** など) を行う場合、アクティブな MQSeries キュー・マネージャーに対する新しい接続が確立され、操作は予測どおりに実行されます。

制御シャットダウン

制御シャットダウンを使って MQSeries キュー・マネージャーを停止しても、接続が即時に切断されることはありません。すべての接続がクローズされるまで待機します (これは、Java バインディングまたは Java クライアント・モードを使って構成される接続に当てはまります)。アクティブな MQSeries-ブリッジ伝送キュー・リスナーは、MQSeries システムが静止していることを通知し、関連する警告を出して停止します。

アクティブになっている MQSeries-ブリッジ・キューは、以下の状態になるまで、MQSeries キュー・マネージャーへの接続を保存します。

- アイドル・タイムアウト期間 (クライアント接続ブリッジ・オブジェクトで指定された期間) の間アイドル状態になった後の接続タイムアウト。この時点で中断した接続はクローズされ、MQSeries キュー・マネージャーの制御シャットダウンは完了します。
- MQSeries-ブリッジ・キューには、中断した接続を使用しようとする何らかのアクション (たとえば、MQSeries に対してメッセージを送る) をとるように通知が出されま

す。 putMessage 操作は失敗し、中断した接続はクローズされます。そして、MQSeries キュー・マネージャーの制御シャットダウンが完了します。

ブリッジ・クライアント接続オブジェクトは接続のプールを保守し、それらは使用されるのを待機します。ブリッジ・アクティビティーがない場合、接続アイドル時間がアイドル・タイムアウト時間 (クライアント接続オブジェクトの構成で指定された時間) を超えるまで、プールには MQSeries クライアント・チャネル接続が保存されます。アイドル・タイムアウト時間を超えた時点で、プール内のチャネルはクローズされます。

MQSeries キュー・マネージャーへの最後のクライアント・チャネル接続がクローズされると、 MQSeries 制御シャットダウンは完了します。

管理オブジェクトとその特性

このセクションでは、MQSeries Everyplace MQSeries-ブリッジに関連したさまざまな管理オブジェクトのタイプについて、それぞれの特性を説明します。特性とは、**inquireAll()** 管理メッセージを使って照会できるオブジェクト属性です。アプリケーションでは結果を読み取って使用することができます。あるいは、特性の値を設定するために管理メッセージの更新または作成時に送信することもできます。特性によっては、管理メッセージの作成および更新を使って設定することもできます。それぞれの特性には固有のラベルが関連付けられており、このラベルを使って特性の値を設定および入手することができます。

以下のリストは、それぞれの管理オブジェクトに当てはまる属性を示しています。属性については、193ページの『属性の詳細』でアルファベット順に詳しく説明されています。ラベル定数は com.ibm.mqe.mqbridge.MQeCharacteristicLabels で定義されます。

ブリッジ・オブジェクトの特性

- Run-state
- Children
- Child

ブリッジ・オブジェクトの特性

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- HeartBeatInterval
- DefaultTransformer

MQSeries キュー・マネージャー・プロキシー・オブジェクトの特性

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- HostName

クライアント接続オブジェクトの特性

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- ClientConnectionName
- Port
- AdapterClass
- MQUserID
- MQPassword
- SendExit
- ReceiveExit
- SecurityExit
- CCSID
- SyncQName
- SyncQPurgerRulesClass
- MaxConnectionIdleTime
- SyncQPurgeInterval

MQSeries 伝送キュー・リスナー・オブジェクトの特性

- Run-state
- Children
- Child
- AdministerObjectClass

- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- ClientConnectionName
- ListenerName
- DeadLetterQName
- ListenerStateStoreAdapter
- UndeliveredMessageRuleClass
- TransformerClass

属性の詳細

属性: AdapterClass

タイプ: Unicode

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_ADAPTER_CLASS`

有効なアクション
照会、作成、更新

説明 これは、Java クラス名か、または Java クラス名に解析できる別名です。ゲートウェイ・スレーブによって使用されます。

デフォルト値は com.ibm.mqe.mqbridge.MQeMQAdapter です。このパラメーターの妥当性検査は行われません。

属性: AdministeredObjectClass

タイプ: Unicode

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_ADMINISTERED_OBJECT_CLASS`

有効なアクション
照会、作成、更新

説明 ブリッジの名前。

有効な文字は、"0〜9"、"A〜Z"、"a〜z" -、.、% /

属性: BridgeName

タイプ: Unicode

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_BRIDGE_NAME`

有効なアクション
照会、作成、更新、削除、開始、停止

説明 記号名を使用する場合、マシンのスイッチが入っていないかを検出したり、ネーム・サーバーが作動していないかを検出したりするのに時間がかかることがあります。これが問題の原因となっている場合は、このフィールドで代わりとして実際の I/P アドレスを使用することができます。

注: 管理メッセージの**「作成」**が使用されているときには、この特性は 1 回しか設定できません。その後、この特性はどのブリッジ管理オブジェクトの照会、更新、削除、開始、または停止を行う必要があるのか識別するために使用されます。

属性: CCSID

タイプ: Int

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CCSID`

有効なアクション
照会、作成、更新

説明 このパラメーターの説明については、MQSeries Java の使用の資料を参照してください。

有効な値は 0 ～ `MAXINT` です。デフォルトは 0 です。

属性: Child

タイプ: Unicode

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CHILD`

有効なアクション
照会

説明 MQSeries-ブリッジ 管理オブジェクトの名前が入っているフィールド。

属性: Children

タイプ: MQeFields 配列

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CHILDREN`

有効なアクション
照会

説明 Child フィールドの配列。それぞれのエレメントには Child 属性が含まれます。

属性: ClientConnectionName

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CLIENT_CONNECTION_NAME

有効なアクション
照会、作成、更新、削除、開始、停止

説明

注: 管理メッセージの**「作成」**が使用されているときには、この特性は 1 回しか設定できません。その後、この特性はどのブリッジ管理オブジェクトの照会、更新、削除、開始、または停止を行う必要があるのか識別するために使用されます。

属性: DeadLetterQName

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_DEAD_LETTER_Q_NAME

有効なアクション
照会、作成、更新

説明 ゲートウェイが MQSeries から MQSeries Everyplace にメッセージを送達できないことを検出する場合、ゲートウェイはメッセージを処理することができません。そのため、そのメッセージは、MQSeries システムの送達不能キューに入れられます。このパラメーターは、エラーが起きているメッセージの送達先のキューを定義します。

デフォルト値は SYSTEM.DEAD.LETTER.QUEUE です。

属性: DefaultTransformer

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_DEFAULT_TRANSFORMER

有効なアクション
照会、作成、更新

説明 ここで指定されるクラス名がデフォルトの変換機能クラスとして使用されます。 MQSeries から MQSeries Everyplace にメッセージが送信されるとき、宛先キューが変換機能クラスを定義する場合があります。定義されない場合、このクラスが MQSeries メッセージを MQSeries Everyplace フォーマットに変換するために使用されます。

MQSeries Everyplace から MQSeries にメッセージが送信されるとき、メッセージを MQSeries Everyplace に移動させる伝送キュー・リスナーに変換機能クラスが定義される場合があります。定義されない場合、このクラスが MQSeries Everyplace メッセージを MQSeries フォーマットに変換するために使用されます。

このフィールドの値の妥当性検査は行われません。

デフォルト値は com.ibm.mqebridge.MQeBaseTransformer です。

属性: HeartBeatInterval

タイプ: Int

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_HEARTBEAT_INTERVAL

有効なアクション
照会、作成、更新

説明 1 分単位で表現した時間間隔 (1 ～ 60 の間の値)。ブリッジは内部的な心拍を使用して、他の管理オブジェクトに定期的な刺激を伝えます。心拍イベント (「クライアント接続は古くなった MQSeries をリープします」や「同期キューは除去されます」など) が届くと、管理オブジェクトは小さいタスクを実行します。心拍は、これ以上小さくできない、タイマーの間隔を提供します。この値が低く設定されると、タイマー関連のアクションはさらに正確になります。たとえば、「アイドル時間が 10 分を超えた場合はすべての MQSeries 接続をリープします」と決定しても、心拍間隔が 3 分に設定されている場合、アイドル状態になっている MQSeries 接続は、3、6、9、12 分後にそれぞれ検査されますが、リープされるのは 12 分後だけです。この値を低く設定すると、タイマー関連の心拍イベントの正確度は高くなりますが、作業効率が犠牲になります。作成される心拍イベントが増えるほど、実行する作業も増えるからです。

デフォルト値は 5 分です。

属性: Hostname

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_HOST_NAME

有効なアクション
照会、作成、更新

説明 MQSeries Java クラスを使ってこの MQSeries キュー・マネージャーへの接続を作成するために使用されます。この特性を指定しない場合、MQSeries キュー・マネージャーは JVM と同じマシン上にあるものと見なされ、 MQSeries システムとの対話には Java バインディング・モードが使用されます。

注: 値をブランクのままにしても、localhost を指定したことにはなりません。ブランク値を使用すると、MQSeries-ブリッジは、MQSeries と直接対話を行う MQSeries Java クラスをバインディング・モードで使用します。localhost を指定する場合、MQSeries-ブリッジは、クライアント・モードで MQSeries Java クラスを使用します。これは、MQSeries とのすべての通信が ネットワーク (TCP/IP) スタックを介して行われることを意味しています。

ここで指定された値の妥当性検査は行われません。記号名を使用すると、マシンのスイッチが入っていないかを検出したり、ネーム・サーバーが作動していないかを検出したりするのに時間がかかることがあります。記号名が問題の原因となっている場合は、このフィールドで I/P アドレス表記を使用することができます。

属性: ListenerName

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_LISTENER_NAME

有効なアクション
照会、作成、更新、削除、開始、停止

説明 このリスナーの名前。リスナー名は、リスナーがメッセージを受け取る MQSeries 上の伝送キューの名前です。 *MQ_queue_manager_name* と *MQ_transmission_queue_name* のペアの組み合わせは、存在するすべてのゲートウェイの間で固有でなければなりません。

注: 管理メッセージの**「作成」**が使用されているときには、この特性は 1 回しか設定できません。その後、この特性はどの MQSeries-ブリッジ管理オブジェクトの照会、更新、削除、開始、または停止を行う必要があるのか識別するために使用されます。

属性: ListenerStateStoreAdapter

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_LISTENER_STATE_STORE_ADAPTER

有効なアクション
照会、作成、更新

説明 永続メッセージを確実に送達するために、リスナー・クラスでは、アダプターを使用して状況情報を保管します。これは、ディスクとの間の状況情報の保管およびリカバリーを管理するためにロードされたアダプターのクラス名 (またはクラス名の別名) です。現在、2 つのアダプターがサポートされています。それは com.ibm.mqe.adapters.MQeDiskFieldsAdapter (ローカル・ファイル・システムに状況情報を保管) と com.ibm.mqe.mqbridge.MQeMQAdapter (MQSeries サ

ーバーに状況情報を保管) です。一般に、ディスク・アダプターを使うと MQSeries ベースのアダプターを使う場合よりも処理速度が速くなります。クラス名の後には、コロンで区切った引き数のリストを続けることができます。ただし、これを使用するのは MQeDiskFieldsAdapter だけです。この場合、MQeDiskFieldsAdapter の後にコロンと、状況情報を含むファイルへの完全修飾パス名を続けることができます。たとえば、ディスク・フィールド・アダプターを使ってリスナーの状況情報をファイル c:¥folder¥state.sta に保管するには、listener-state-store-adapter フィールドに `com.ibm.mqe.Adapters.MQeDiskFieldsAdapter:c:¥folder¥state.sta` という値が含まれていなければなりません。このパラメーターで指定されたファイルは、現在存在している必要はありません。提供されたパス名がフォルダー区切り文字 (たとえば、DOS では "¥") で終わっている場合には、提供されたパラメーターがディレクトリーであると想定されます。そして、<ListenerName>-listener.sta という状況ファイルがそのディレクトリーに作成されます (ここで、<ListenerName> はレジストリー項目にあるリスナーの名前です)。パス名が提供されていない場合、リスナーは現行の Java 作業ディレクトリーにある <ListenerName>-listener.sta というファイルを使用します。MQeMQAdapter が使用されている場合、追加の引き数を指定する必要はありません。

ListenerStateStoreAdapter フィールドのデフォルト値は `com.ibm.mqe.Adapters.MQeDiskFieldsAdapter` です。

属性: MaxConnectionIdleTime

タイプ: Int

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_MAX_CONNECTION_IDLE_TIME`

有効なアクション

照会、作成、更新

説明 ブリッジにあるそれぞれのクライアント接続オブジェクトは、その MQSeries システムへの MQSeries Java クライアント接続のプールで保守されます。

MQSeries 接続が使用されずにアイドル状態になると、タイマーが開始されます。タイマーがこのパラメーターの現行値になると、アイドル状態の接続はスローされます。これを接続のリープ と言います。これにより、接続がアイドル状態になるときにリソースが節約されます。接続プールは、MQSeries-ブリッジで使用される有用なデバイスです。新しい MQSeries クライアント接続を作成することはリソースを集中的に用いる操作です。プール内にアイドル状態の接続がある場合、その 1 つを再利用すると、新規接続の作成操作を行わずに済みます。 *MaxConnectionIdleTime* の値が高くなると、アイドル状態の接続が接続プールで待機している可能性も大きくなります。しかし、何も実行しないで JVM 内のリソースを消費するクライアント接続が増えます。この値を低く設定

すると、アイドル状態の接続が使用可能になっている可能性は小さくなりますが、それと同時にアイドル状態の接続の数も少なくなります。そのため、消費されるリソースが減ります。

時間は 1 分単位で表現されます。

有効な範囲: 0 ～ 720 の間 (12 時間)。デフォルト値は 5 (分) です。

この値を 0 に設定すると、接続プールを使用しないことを意味し、 MQSeries クライアント接続がアイドル状態になっているときには、リープまたは破棄が行われます。

このタイムアウトは、 *heartbeatInterval* パラメーターで設定される間隔でチェックされるに過ぎません。

MaxConnectionIdleTime は、MQSeries Everyplace システムのシャットダウンにかかる時間に直接的な影響を与えることがあります。詳細については、189ページの『MQSeries キュー・マネージャーのシャットダウンの際の MQSeries-ブリッジの考慮事項』を参照してください。

属性: MQPassword

タイプ: Unicode

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_PASSWORD`

有効なアクション
照会、作成、更新

説明 MQSeries Java クラスによって使用されます。この属性が指定されていない場合、 MQSeries 呼び出しのパスワード・フィールドは、″ ″ (ブランク) に設定されます。ここで指定する値は、使用されているデフォルトをオーバーライドします。このパラメーターの妥当性検査は行われません。

属性: MQQMgrProxyName

タイプ: Unicode

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_MQ_Q_MGR_PROXY_NAME`

有効なアクション
照会、作成、更新、削除、開始、停止

説明 キュー・マネージャー・プロキシー・オブジェクトの名前。 (つまり、宛先 MQSeries キュー・マネージャーの名前。)

注: 管理メッセージの**「作成」**が使用されているときには、この特性は 1 回しか設定できません。その後、この特性はどのブリッジ管理オブジェクトの照会、更新、削除、開始、または停止を行う必要があるのか識別するために使用されます。

属性: MQUserID

タイプ: Unicode

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_USER_ID`

有効なアクション
照会、作成、更新

説明 MQSeries Java クラスによって使用されます。このパラメーターが指定されていない場合、 MQSeries 呼び出しのユーザー ID フィールドは、" " (ブランク) に設定されます。ここで指定する値は、使用されているデフォルトをオーバーライドします。このパラメーターの妥当性検査は行われません。

属性: Port

タイプ: Int

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_PORT`

有効なアクション
照会、作成、更新

説明 MQSeries Java クラスを使ってこの MQSeries キュー・マネージャーへの接続を作成するために使用されます。このパラメーターを指定しない場合、 MQSeries キュー・マネージャーは JVM と同じマシン上にあるものと見なされ、 MQSeries システムとの対話には MQSeries Java クラスのバインディング・モードが使用されます。

有効な範囲は 0 ～ `MAXINT` です。

属性: ReceiveExit

タイプ: Unicode

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_RECEIVE_EXIT`

有効なアクション
照会、作成、更新

説明 クライアント・チャネルの他方の終端で使われる出口を突き合わせるために使用されます。

このパラメーターの妥当性検査は行われません。

属性: Run-state

タイプ: Int

ラベル:

com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_RUN_STATE

有効なアクション
照会

説明 管理オブジェクトが実行中つまり使用されているか (値 =1)、それとも停止つまり使用されていないか (値 =0) を示します。オブジェクトが停止されるときには、そのプロパティーを変更することができます。

属性: SecurityExit

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SECURITY_EXIT

有効なアクション
照会、作成、更新

説明 クライアント・チャネルの他方の終端で使われる出口を突き合わせるために使用されます。

このパラメーターの妥当性検査は行われません。

属性: SendExit

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SEND_EXIT

有効なアクション
照会、作成、更新

説明 クライアント・チャネルの他方の終端で使われる出口を突き合わせるために使用されます。

このパラメーターの妥当性検査は行われません。

属性: StartupRuleClass

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_STARTUP_RULE_CLASS

有効なアクション
照会、作成、更新

説明 * これは、管理オブジェクトがシステム始動時または最初の作成時にロードされたときに使用されるルール・クラスです。ルール・クラスの名前の妥当性検査は行われません。

ルール・クラスは、管理オブジェクトが開始されているか、およびその子が開始されているかどうかを示します。デフォルトのルールは com.ibm.mqe.mqbridge .MQeStartupRule です。このデフォルトを指定すると、管理オブジェクトが開始され、そのすべての親が始動します。このフィールドが ″ ″ (ブランク) に設定される場合、管理オブジェクトは開始されません。215ページの『MQeStartupRule』を参照してください。

属性: SyncQName

タイプ: Unicode

ラベル:
`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SYNC_Q_NAME`

有効なアクション
照会、作成、更新

説明 この MQSeries キュー・マネージャーで MQSeries-ブリッジによって使用される同期キューの名前。有効な文字は、"0〜9"、"A〜Z"、"a〜z"、_、.、%、/ です。同期キューとは、メッセージを MQSeries Everyplace から MQSeries に移動させるプロセスにおいて、メッセージの追跡のために使用される MQSeries キューです。メッセージが
1 回限りのメッセージ送達を保証する処理の途中にある場合、そのメッセージが論理のどの程度まで処理されているかを示す別のメッセージが同期キューにあります。 MQSeries Everyplace システムが完全にシャットダウンされる場合、同期キューは空になっているはずです。システム間の接続が破損した場合、一部の永続状況情報は同期キューに残されます。 MQSeries Everyplace システムは、再始動して、プロセスが失敗した場所から続行するときに、この情報を使用します。同期キューの名前は、同じブリッジ上のクライアント接続については同じものにすることができます。また、その同期キューとの対話時に使用される送信、受信、セキュリティー出口が同じ場合は、別のブリッジ上にあっても同じ名前にすることができます。同期キューは、MQSeries Everyplace -> MQSeries へのメッセージ転送を処理するために、MQSeries キュー・マネージャー上になければなりません。リスナー状態クラスが MQeMQAdapter の場合、その同期キューがリスナーに関する永続状況情報を保管するためにも使用されることを意味します。状況情報が MQeDiskFieldsAdapter で保管されている場合、リスナーはこのパラメーターを使用しません。どのクライアント接続がどの同期キューを使用するかを理解しておくために、`MQE.SYNCQ.<ClientConnectionName>` という命名体系を使用することをお勧めします。

デフォルト値は `MQE.SYNCQ.DEFAULT` です。

属性: SyncQPurgeInterval

タイプ: int

ラベル:

com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SYNC_Q_PURGE_INTERVAL

有効なアクション
照会、作成、更新

説明

属性: SyncQPurgerRulesClass

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SYNC_Q_PURGER_RULES_CLASS

有効なアクション
照会、作成、更新

説明 同期キューにあるメッセージが MQSeries Everyplace の障害を示すときに、メッセージを確認するために使用されるルール・クラスの名前。

デフォルトは、MQSeries Everyplace トレースの状態を報告するクラス名です。

このパラメーターの妥当性検査は行われません。

属性: TransformerClass

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_TRANSFORMER

有効なアクション
照会、作成、更新

説明 これは、MQSeries メッセージを MQSeries Everyplace メッセージに変換するために使用される Java クラスの名前です。メッセージがリスナーによって MQSeries から取られると、指定された変換機能を使って MQSeries Everyplace フォーマットのメッセージに変換されます。変換機能クラスが null またはブランクのストリングに指定されている場合、ブリッジ構成パラメーターで提供された *DefaultTransformer* パラメーターが変換機能として使用されます。デフォルトも null またはブランクに設定されている場合、メッセージは転送されません。

デフォルト値は ″ ″ (ブランク) です。

詳細については、208ページの『変換機能』を参照してください。

属性: UndelivedMessageRuleClass

タイプ: Unicode

ラベル:
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_UNDELIVERED_MESSAGE_RULE_CLASS

有効なアクション

照会、作成、更新

説明 MQeUndeliveredMessageRule クラスの名前。 MQSeries から MQSeries Everyplace に移動されるメッセージを送達できないと、このルール・クラスが調べられ、リスナーがとるべきアクションが決定されます。リスナーに通知されるアクションは、待機および再試行、シャットダウン、または MQMessage レポート・オプションで定義されたメッセージ処理です。

デフォルト値は `com.ibm.mqe.mqbridge.MQeUndeliveredMessageRule` です。
214ページの『MQeUndeliveredMessageRule』を参照してください。

メッセージを MQSeries から MQSeries Everyplace に送信する方法

メッセージの伝送をテストするために、MQSeries システムで経路指定を調整するには、多くの方法があります。 1 つの方法は、ブリッジ・セットアップ・ウィザード・ツール (187ページの『管理 GUI アプリケーションの例』に説明されているように) を使って、認知されている MQSeries Everyplace キュー・マネージャーごとにキュー・マネージャー別名を定義することです。本書では、MQSeries Everyplace キューにメッセージを送信するために、結果の構成を使用する方法を説明します。

1. MQSeries クライアント v 5.1 から、MQSeries First Steps プログラムを選択する
2. 「最初のステップ (First Steps)」画面から、**「API エクササイザー」**を選択する
3. 「API エクササイザー・キュー・マネージャー (API Exerciser Queue Managers)」画面で次のことを実行する
   - ブリッジの接続先の MQSeries キュー・マネージャーを選択する (例では `MQA`)
   - **「拡張モード (Advanced mode)」**チェック・ボックスをチェックする
   - **「MQCONN」**ボタンをクリックする
   - **「キュー (Queues)」**タブを選択して「キュー (Queues)」画面を表示する
   - **「MQOPEN」**を選択して「MQOPEN 選択可能オプション (MQOPEN Selectable Options)」画面を表示する
4. 「MQOPEN 選択可能オプション (MQOPEN Selectable Options)」画面で次のことを実行する
   - **「MQOO_INPUT_AS_Q_DEF」** が選択されていないことを確認する
   - **「MQO_OUTPUT」**が選択されていることを確認する
   - メッセージの宛先にする MQSeries Everyplace キュー・マネージャーで、**「ObjectName」**フィールドにキューの名前を入力する (例では `Q1`)
   - メッセージの宛先にする MQSeries Everyplace キュー・マネージャーの名前を、**「ObjectQMgrName」**フィールドに入力する (例では `ExampleQM`)
   - **「OK」**をクリックしてキューへの経路をオープンする
5. 「API エクササイザー・キュー (API Exerciser Queues)」画面で次のことを実行する

- **「MQPUT」**ボタンを選択して「MQPUT - 引き数オプション (MQPUT - Argument Options)」画面を表示する

6. 「MQPUT - 引き数オプション (MQPUT - Argument Options)」画面で次のことを実行する
   - メッセージを入力する
   - **「OK」**をクリックしてメッセージを MQSeries Everyplace システム上の `ExampleQM` の `Q1` に送信する

配布不能メッセージの処理

MQSeries-ブリッジの伝送キュー・リスナーは MQSeries チャネルに似ており、MQSeries 伝送キューからメッセージをプルしたり、メッセージを MQSeries Everyplace ネットワークに送達することができます。これは MQSeries Everyplace ルールに準拠しており、メッセージが送達できない場合は、*未配布メッセージ・ルール* を参照して伝送キュー・リスナーがどのように反応すべきかを判別します。ルールがメッセージ・ヘッダーのレポート・オプションを指し示し、レポート・オプションがメッセージを送達不能キューに書き込むよう指示すると、メッセージは MQSeries 送達不能キューに (*送達側*のキュー・マネージャーに) 入れられます。

MQSeries-ブリッジ キューへのメッセージの書き込み

アプリケーションが **putMessage** を使用しており、 **confirmputMessage()** を使ってこのメッセージを確認してはならないことを指定している場合、 MQSeries-ブリッジはメッセージを MQSeries に渡すための確実な送達論理を使用しません。これは、宛先 MQSeries キューへの単純な **MQPut** を行います。メッセージの経路のどこかで障害が起きた場合には、アプリケーションはメッセージが送信されたかどうかを判別することはできません。アプリケーションがメッセージを再送信することを決定する場合、 2 つの同一のメッセージが MQSeries キューに届く可能性があります。

この問題を回避するために、アプリケーション・プログラマーは、 **putMessage()** と **confirmputMessage()** 呼び出しの組み合わせを使用する必要があります。 *confirm* パラメーターで `true` を指定した **putMessage()** を使用すると、 MQSeries-ブリッジは確実な送達論理を使用して、メッセージを MQSeries システムに書き込みます。

MQSeries システムと送信側アプリケーションとの間のパスのコンポーネントに障害が発生した場合、アプリケーションは、メッセージが送信先に達したかどうかを判別することができなくなります。この場合、アプリケーションは再び元のメッセージを取って、それにブール MQeField を追加する必要があります。たとえば、次のようにします。

```
msg.putBoolean( MQe.Qos_Retry)
```

これは、このメッセージが以前に送信されていることを示しています。 MQSeries-ブリッジはその確実な送達論理を使用して、2 つの **putMessage()** 呼び出しのうちの 1 つだけが、実際にメッセージを MQSeries に書き込むようにすることができます。

*確認*フラグを設定しないで **putMessage()** を使用し、正常な戻りコードを受け取った場合には、アプリケーションはメッセージが MQSeries キューに渡されたことを確証できたことになります。

確認 フラグを設定して **putMessage()** を使用した場合、 MQSeries-ブリッジはメッセージに関するいくらかの情報を (同期キューに) 保持して、アプリケーションが同じメッセージを重複して送信しないようにすることができます。 *Qos_Retry* パラメーターが設定される場合に限って、 MQSeries-ブリッジは、同じメッセージを重複して送信しないようにすることができます。 **confirmputMessage()** は、 MQSeries-ブリッジ同期キューからメッセージ・ヒストリーを除去します。

次の手順では、4 つのメッセージが宛先 MQSeries キューに置かれます。

	新規メッセージの作成	
(1)	`putMessage(Confirm=Yes)`	- メッセージは MQSeries に送達されるが、何らかの注釈が同期キューに作成される
	メッセージ上に再試行ビットが設定される	
	`putMessage(Confirm=Yes)`	- メッセージの注釈がすでに同期キューにあると、抑止される
	`putMessage(Confirm=Yes)`	- メッセージの注釈がすでに同期キューにあると、抑止される
(2)	`putMessage(Confirm=No)`	- メッセージが MQSeries キューに送達されると、抑止されない
	メッセージから再試行ビットを除去する	
(3)	`putMessage(Confirm=Yes)`	- メッセージが MQSeries に送信される。再試行ビットが設定されておらず、 MQSeries-ブリッジはその同期キューを調べなかった
	`ConfirmputMessage()`	- MQSeries-ブリッジがそのメッセージ・メモリーを消去する
	メッセージ上に再試行ビットが設定される	
(4)	`putMessage()`	- メッセージが送信される。

MQSeries-ブリッジ キューからのメッセージの取得およびブラウズ

他の MQSeries Everyplace キューの場合と同様に、MQSeries-ブリッジ キューからメッセージを取得およびブラウズすることができます。また、これらの操作で MQeFields フィルターを指定することも可能です。フィルターを使用する場合、そのフィルターと一致する最初のメッセージが **getMessage** によって戻され、そのフィルターと一致するすべてのメッセージが **browseMessages** によって戻されます。

メッセージをブラウズする場合、フィルター・フィールドがブランクまたはヌルであれば、すべてのメッセージが MQSeries キューから収集されて、戻りの列挙に入れられま

す。フィルターが非ブランクまたは非ヌルであれば、 MQSeries キューから収集されたすべてのメッセージがキューのメッセージ変換プログラムに渡されて、フィルターと突き合わされます。一致するメッセージは戻りの列挙に入れられます。

フィルター・フィールドに *Msg_MsgID* または *Msg_CorrelId* (あるいはその両方) が含まれている場合は、MQSeries *メッセージ Id* または *CorrelId* (あるいはその両方) をフィルター素子として使用して MQSeries からメッセージを収集します。次に、その結果が MQSeries Everyplace メッセージに変換されます。それらのメッセージは、以下のようにしてフィルターに掛けられます。

1. オリジナルのフィルターがデフォルトの一致基準として適用され、一致するすべてのメッセージが戻りの列挙に入れられます。
2. 変換済みのいずれかの MQSeries Everyplace メッセージに *Msg_MsgID* フィールドが含まれていなければ、*Msg_MsgID* フィールドがフィルターから除去されます。
3. 変換済みのいずれかの MQSeries Everyplace メッセージに *Msg_CorrelId* フィールドが含まれていなければ、*Msg_CorrelId* フィールドがフィルターから除去されます。
4. 次に、一致しない *MQSeries Everyplace* メッセージが新しいフィルターに掛けられ、一致するメッセージが戻りの列挙に入れられます。

ブランク・フィルターまたはヌル・フィルターを使用したり、*Msg_MsgID* フィールドも *Msg_CorrelId* フィールドも含まれていないフィールドを使用すると、MQSeries キューのすべてのメッセージがブラウズされます。最適パフォーマンスを得るためには、MQSeries フォーマットになった *メッセージ Id* または *CorrelId* (あるいはその両方) をフィルターに含めるようにします。

getMessage のフィルターは、 **browseMessages** のフィルターと同じように動作します。ただし、最初に一致したものだけが MQSeries キューから除去されて、アプリケーションに戻されます。

使用上の制約事項

getMessage および **browseMessages** を MQSeries-ブリッジ キューで使用する場合は、以下のようないくつかの制約事項が適用されます。

- MQSeries リモート・キュー定義をポイントする MQSeries-ブリッジ キューからメッセージを取得またはブラウズすることはできません。
- MQSeries-ブリッジ キュー取得に非ゼロ *確認 ID* を使用することはできません。つまり、 MQSeries-ブリッジ キューに関する **getMessage** 操作では確実な送達は提供されません。取得操作を確実に行う必要がある場合は、転送キュー・リスナーを使用して MQSeries のメッセージを転送する必要があります。
- MQSeries から出るメッセージには固有 ID が含まれていないため、真に設定された *justUID* フラグを使用してメッセージをブラウズすることはできません (この操作は、通常、ブラウズと一致する固有なメッセージ ID のリストを戻します)。
- **browseMessagesAndLock()** メソッドはサポートされません。

変換機能

変換機能は、MQSeries Everyplace メッセージを MQSeries メッセージに変換したり、MQSeries メッセージを MQSeries Everyplace メッセージに変換したりできる Java クラスです。変換機能は MQeBaseTransformer クラスから派生します。

変換機能は、MQSeries-ブリッジ構成中にいくつかの方法で指定できます。

- デフォルト変換機能は各 MQSeries-ブリッジごとに指定できます。
- 変換機能は各 MQSeries-ブリッジ・キューごとに指定できます。
- 変換機能は各 MQSeries 伝送キュー・リスナーごとに指定できます。

変換機能は、メッセージ変換のすべての局面を扱うので、使用したい MQSeries および MQSeries Everyplace メッセージ・フォーマットの間の変換のメソッドを備えていなければなりません。つまり、MQSeries および MQSeries Everyplace の間を流れるメッセージに新しいフォーマットを作成する場合はいつでも、その新しいメッセージ・フォーマットに変換機能クラスを作成するか、変更する必要があります。

これらの変更は、さまざまな方法で扱うことができます。

- すべてのメッセージ・フォーマットを変換できる、単一の巨大な変換機能を作成する。

  これは、ある変換機能が別の機能から継承し、その機能が別の機能から継承するというようにして、変換機能のチェーンを形成する、Java の継承モデルを使用して実現されます。または、1 つの Java クラスとして実現されます。

  このアプローチの**利点**は次のとおりです。

  – この変換機能は、MQSeries-ブリッジのデフォルトとして指定できます。これには、すべての操作に使用する変換機能を判別するために、構成のポイントを 1 つだけ必要とします。 (MQSeries-transmission-queue-listener および MQSeries-ブリッジ・キュー定義で、変換機能名をブランクまたは `null` のままにしておきます。)

  – 非常に単純なアプローチです。

  このアプローチの**欠点**は次のとおりです。

  – アプリケーションのフォーマットの変更時、または新規フォーマットの作成時に、この大きな変換機能をすべての箇所で変更し、再展開する必要があります。

  – システム中のすべてのメッセージ・フォーマットを理解する、 1 つの変換機能を作成することは不可能である場合があります。

- それぞれがさまざまなグループのメッセージ・フォーマットを理解し変換できる、中サイズの一連の変換機能を作成する。

  各変換機能は、特定のアプリケーションを処理する役割を果たし、MQSeries Everyplace 経路指定は、各アプリケーションが MQSeries-ブリッジ・キューのセット、および MQSeries 伝送キュー・リスナーを排他的に使用するように設定できま

す。 MQSeries-ブリッジ・キューおよび伝送キュー・リスナー上の変換機能名は、その後アプリケーション固有に設定されます。

このアプローチの**利点**は次のとおりです。

- プログラマーはメッセージの経路指定される位置を完全に制御でき、正しい変換機能が使用されることを確認できます。
- アプローチが単純です。
- メッセージ・フォーマットを追加または変更する場合、変換機能は変更済みの、または新しいメッセージのフローの経路に沿って変更するだけですみます。

- システムのメッセージ・フォーマットごとに個別の変換機能を作成する。

  これには、非常に小さな変換機能のリストを使用する、高レベルの変換機能を作成することが必要です (210ページの『examples.mqbridge.transformers.MQeListTransformer 変換機能クラスの例』を参照)。この機能は、メッセージを使用できる変換機能が見つかるまで、それぞれを順番に呼び出します。

  各変換機能は、メッセージ・フォーマットを 1 つずつ理解しています。

  各メッセージ・フォーマットに注意を払い、さらに、小さな変換機能のそれぞれが変換するメッセージのフォーマットを固有に識別することを確認するため、変換機能にも注意する必要があります。メッセージのインスタンスが、複数の変換機能によって変換できるようにならないようにしてください。各変換機能は各メッセージを調べて、メッセージが、変換機能が処理するように設計されたフォーマットに適合しているか判別できなければなりません。

  さまざまなリストの変換機能が、MQSeries-ブリッジ構成の異なるポイントで使用されることがあります。最も基本的なレベルでは、使用可能な小さな変換機能全体のリストで、リスト変換機能を作成し、これをデフォルトに設定します。最も複雑なレベルでは、変換機能の非常に小さなリストでリスト変換機能を作成し、 MQSeries-ブリッジ・キューおよび MQSeries 伝送キュー・リスナーの変換機能パラメーターを設定します。

  リスト変換機能は、次のいずれかからそのリストを入手することができます。

  - Java ソース・コード自体の中にあるハードコーディングされたリテラル・ストリング定数
  - JVM のシステム環境変数
  - 基礎となっているオペレーティング・システム環境
  - リスト変換機能クラスがロードされるときにロードされる ASCII データ・ファイル
  - ロードされるときに、ファイル・システム内で使用可能なのはどの変換機能クラスであるかを調べることによって

  メソッドの選択は、アプリケーション・プログラマーに任されます。リスト変換機能のサンプルでは、Java ソース・コード内の変換機能リストのハード・コーディングのメソッドを使用します。

このアプローチの利点は次のとおりです。

- このアプローチは、よりオブジェクト指向であり、単一の小さな変換機能内で完全にカプセル化される、特定のメッセージ・フォーマットについて知ることができますが、リスト変換機能は、使用可能な変換機能を理解するだけです。
- 小さな変換機能を新しく追加しても、リスト変換機能を変更する必要はありません。たとえば、リスト変換機能がファイル・システムを探してどの変換機能が使用可能かを調べる場合、単にファイル・システムの正しい位置に変換機能を追加するだけで、その変換機能が使用されるようにできます。

• 上記のメソッドをすべて混合して使用する。

注: *Msg_CorrelID* または *Msg_MsgID* フィールドのいずれかが含まれた MQSeries Everyplace フォーマットのメッセージが変換機能によって戻された場合、それらの内容は、オリジナルの MQSeries スタイル・メッセージと完全に一致していなければなりません。この規則は MQSeries-ブリッジ キューによって強制されます。このガイドラインに違反した場合は、戻された MQSeries Everyplace メッセージがブラウズ・アプリケーションに見えなくなります。

examples.mqbridge.transformers.MQeListTransformer 変換機能クラスの例

このサンプル変換機能では、それ自体はメッセージのフォーマットを理解していなくても、メッセージの変換を実行するために、高レベルの変換機能クラスが非常に小さい変換機能のリストを使用する方法を示します。

ソース・ファイルは examples¥mqbridge¥transformers¥MQeListTransformer.java で、単純な MQSeries から MQSeries Everyplace への変換機能クラスです。

この変換機能は、渡されるメッセージのフォーマットを理解していません。小さい変換機能の番号付きリストがあります。メッセージを変換する必要がある場合、このクラスは変換機能のリスト中を 1 つずつ処理し、各変換機能にメッセージを提示します。最初の変換機能が変換されたメッセージを正常に戻した結果は、このクラスのユーザーに戻されます。

このスタイルの変換機能は、それぞれの小さい変換機能が限られた数のメッセージ・フォーマットを理解している、小さい変換機能の集合と結び付けて使用できます。

このクラスは、自分の変換機能のリストを、静的番号付きリスト (配列) の中に保持します。

サンプルを使用するには、一連の小さい変換機能を作成し、それらのクラス名をサンプル・ファイルの最上部の静的リストに入れます。変更されたサンプル変換機能を、MQSeries-ブリッジ構成の必要な場所にコンパイルし、設定します (175ページの『基本インストールの構成』を参照)。

Msg_CorrelID または *Msg_MsgID* フィールドのいずれかが含まれた MQSeries Everyplace フォーマットのメッセージを変換機能が展開する場合は、それらの内容は、オリジナルの MQSeries スタイル・メッセージの *メッセージ ID* および *相関 ID* と一致しなければなりません。このガイドラインに違反すると、アプリケーションが、*Msg_CorrelID* または *Msg_MsgID* のいずれかが含まれたフィルターを使用してブリッジ・キューをブラウズするときに、変換済みのメッセージがブラウズ・アプリケーションに見えなくなります。

MQSeries スタイル・メッセージ

MQeMQMsgObject は、MQSeries Everyplace 内の MQSeries スタイル・メッセージをサポートする MQeMsgObject のサブクラスです。これは、通常、MQSeries-ブリッジでデフォルトの変換機能を使用して MQSeries アプリケーションとメッセージを交換するのに使用されます。デフォルトの変換機能は、標準 MQSeries メッセージを受け取る際に、MQeMQMsgObject を生成します。同様に、MQSeries Everyplace アプリケーションが MQeMQMsgObject を生成して MQSeries に送信すると、ブリッジにあるデフォルトの変換機能は、これを標準 MQSeries メッセージに変換する方法を認識します。

MQeMQMsgObject クラスがご使用の要件に適合しない場合には、ご使用のアプリケーションに適合する別のタイプのメッセージ・オブジェクトを使用するブリッジで、変換機能を作成することができます。

MQSeries スタイル・メッセージの読み取り

アプリケーションがメッセージを受け取ると、以下のようにしてこのメッセージが MQeMQMsgObject クラスのものであるかどうかを検査できます。

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
  MQeMsgObject msg = MyQM.getMessage(qmgr, queue, null, null, 0);
  if (msg instanceof MQeMQMsgObject)
  {
    MQeMQMsgObject mqeMsg = (MQeMQMsgObject) msg;
    ...
  }
```

メッセージがこのクラスのものである場合、メッセージ・オブジェクトで適切な**取得**メソッドを使用することにより、メッセージ・データと同様に、MQSeries メッセージ・ヘッダーからのすべての情報にアクセスできます。ヘッダー情報は、形式 **get***xxx* **()** のメソッドを使用して取得できます。ここで、*xxx* は、ヘッダー・フィールドの名前です。整合性を守るために、ヘッダー・フィールドの名前とタイプは、MQSeries Java クラスの名前とタイプに一致したものになります。アプリケーション・データは、**getData()** メソッドを使用して取得されます。

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
  if (msg instanceof MQeMQMsgObject)
  {
    MQeMQMsgObject mqeMsg = (MQeMQMsgObject) msg;
```

```
    String replyQMgr = mqeMsg.getReplyToQueueManagerName();
    String replyQueue = mqeMsg.getReplyToQueueName();
    byte [] correlId = mqeMsg.getCorrelationId();
    String msgFormat = mqeMsg.getFormat();
     ...
    byte [] data = mqeMsg.getData();
     ...
  }
```

これで、データはアプリケーションによって処理されます。 MQeMQMsgObject は、データをバイト配列として戻し、アプリケーションは、バイト配列内でデータの構造を理解しなければなりません。さらに構造化されたフォーマットのデータが必要な場合、アプリケーション・データを理解し、これを必要なフォーマットに変換する変換機能を自分自身で作成することができます。

MQSeries スタイル・メッセージの作成

デフォルトの変換機能が理解できる MQSeries スタイル・メッセージを作成するには、新しい MQeMQMsgObject を作成し、ヘッダー・フィールドおよびデータに必要な値を設定します。通常の方法でメッセージを送信します。

新しいメッセージ・オブジェクトを作成するには、コンストラクターを呼び出します。これにはパラメーターはありません。

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
try {
  MQeMQMsgObject mqeMsg = new MQeMQMsgObject()
  ...
}
```

set*xxx* **()** という形式のメソッドを使用して、メッセージ内に MQSeries ヘッダー情報を設定します。ここで、*xxx* はヘッダー・フィールドの名前です。整合性を守るために、ヘッダー・フィールドの名前とタイプは、MQSeries Java クラスの名前とタイプに一致したものになります。明示的に設定されないヘッダー・フィールドは、MQSeries デフォルト値を使用します。

アプリケーション・データは、**setData()** メソッドを使用して設定されます。

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
try {
  MQeMQMsgObject mqeMsg = new MQeMQMsgObject()
  mqeMsg.setPutApplicationName("myApp");
  mqeMsg.setFormat(...);
  mqeMsg.setData(...);
  MyQM.putMessage(qmgr, queue, mqeMsg, null, 0);
}
```

setData() に渡される前に、データは受信側アプリケーションが理解するバイト配列にフォーマットされる必要があります。

変換機能と満了時間の考慮事項

MQSeries と MQSeries Everyplace の間で満了時間を変換する場合は、特別な注意が必要です。

MQSeries Everyplace 満了時間は、メッセージの有効期限が切れた後の明示的時間か、またはメッセージ作成時刻からメッセージが期限切れになるまでの時間の長さを 1 ミリ秒単位で表したデルタのどちらかとして指定されます。

MQSeries 単位は、1/10 秒です。

変換機能がこれらの満了時間を変換できない場合、メッセージの期限が切れ、メッセージは確実に消失します。

MQSeries-ブリッジのルール

MQSeries-ブリッジは以下のようなルール・クラスを使用します。これらのルールによって、ブリッジの動作を変えることができます。

MQeLoadBridgeRule
このルール・クラスは、サーバーの開始時にどのブリッジをロードするかを決定します。

MQeUndeliveredMessageRule
このルール・クラスは、MQSeries Everyplace に送信できない MQSeries メッセージの処理方法を決定します。

MQeSyncQueuePurgerRule
このルール・クラスは、MQSeries Everyplace から MQSeries への古い未確認メッセージに対する処置を決定します。

MQeStartupRule
このルール・クラスは、管理オブジェクトが最初にロードされた時点で、このオブジェクトを開始するかどうかを決定します。

これらのクラスについて、以下のセクションでさらに詳しく説明します。プログラマーは、これらのルール・クラスをサブクラス化し、MQSeries Everyplace の動作を変えるルールを作成して MQSeries Everyplace の構成を変更することによって、デフォルト・ルール・クラスの代わりに独自のルール・クラスを使用することができます。

MQeLoadBridgeRule

このクラスは、サーバー開始時にどのブリッジをロードするかを定義します。サーバーが **MQeMQBridge.activate()** メソッドを使用する場合には、ブリッジ・ローダーが開始します。ブリッジ・ローダーはレジストリーのすべての項目を読み取り、このルール・クラスに従って、レジストリー内の各ブリッジ名ごとに、ブリッジをロードするかどうか判断します。基本的な MQeLoadBridgeRule クラスは、レジストリー内のすべて

のブリッジをロードさせます。単一の MQSeries Everyplace キュー・マネージャーがレジストリーを使用している限り、このようなルールが適切です。

レジストリーが複数の MQSeries Everyplace キュー・マネージャーによって共用されている場合、同じブリッジ・オブジェクトを複数のキュー・マネージャーがロードしようとすることがありますが、これは無効です。すべてのブリッジ、キュー・マネージャーおよびキューへのアクセスは、最初に開始したサーバーに対して許可され、その後で開始したサーバーはすべてロックアウトされます。このため、MQeLoadBridgeRule のカスタマイズ・バージョンを作成することによって、各サーバーがロードするブリッジを選択しておくのが適切です。特定のブリッジをロードする必要のあるサーバーの名前とそのブリッジ名とが関連するような命名規則を使用すれば、カスタマイズ・ルールの作成が容易になります。

クラス examples.mqbridge.rules.ExampleLoadBridgeRule は、ブリッジ・オブジェクトに命名規則を適用する方法を示します。特に LoadBridgeRule と関連付けて使用することによって、サーバーがどのブリッジをロードするかを記述する方法を示します。

MQeUndeliveredMessageRule

MQeUndeliveredMessageRule

1 つのブリッジに対して複数の MQSeries 伝送キュー・リスナー・オブジェクトが定義され、それらが実行されて MQSeries 伝送キューから MQSeries Everyplace ネットワークに一連のメッセージを移動する場合があります。

ある MQSeries メッセージを MQSeries Everyplace ネットワークに送信できない場合、伝送キュー・リスナーのスレッドは**許可**メソッドを呼び出して、リスナーの構成パラメーターに示された UndeliveredMessageRule クラスを調べます。このメソッドからの戻り値は、行うべき処置を示します。

- 結果が値 `MQeUndeliveredMessageRule.STOP_LISTENER` であれば、このメッセージは送信不能のため、リスナーは停止しなければなりません。メッセージは MQSeries システム内の伝送キューに残ります。
- 結果が値 `MQeUndeliveredMessageRule.USE_MQ_REPORT_OPTIONS` であれば、元の MQSeries メッセージの*「レポート (report)」*フィールドの値に応じて、メッセージを廃棄または MQSeries Everyplace システムの送達不能キューに移動する必要があります。 MQSeries キュー・マネージャーの送達不能キューの名前は、伝送キュー・リスナーのMQSeries-ブリッジの構成パラメーターが示しています。この値が戻され、しかもメッセージの*レポート*・オプションが `MQRO_DISCARD` である場合には、未配布メッセージは廃棄されます。
- "0" より大きい整数値が結果として戻された場合、その整数値は、リスナーが MQSeries から MQSeries Everyplace への転送操作を再試行するまでの待機時間を表します。

上のいずれにも該当しない値が戻された場合、またはルールが例外を報告した場合には、リスナーは結果 STOP_LISTENER が戻された場合と同じ動作をします。

クラス examples.mqbridge.rules.MQeUndeliveredMessageRule は、 MQSeries-ブリッジ構成が使用するデフォルト・ルールの動作を示します。このクラスが呼び出されると、失敗が連続する場合に以下の動作が行われます。

- 最初の 1 分間は、再試行まで 5 秒間待機する。
- 次の 1 分間は、再試行まで 10 秒間待機する。
- 2 分経過後から 10 分経過までの間は、再試行まで 60 秒間待機する。
- 10 分経過しても再試行が失敗する場合には、STOP_LISTENER を適用する。

伝送キュー・リスナーの動作を調整するためのクラスのもう 1 つの例に、examples.mqbridge.rules.UndeliveredMQMessageToDLQRule があります。 **permit()** メソッドによって、常に値 MQeUndeliveredMessageRule.USE_MQ_REPORT_OPTIONS が戻されます。

MQeSyncQueuePurgerRule

*同期*キュー は MQSeries キュー・マネージャー上にローカルに定義されたキューで、MQSeries-ブリッジによって排他的に使用され、確実なメッセージ送信のために使用されます。 MQSeries あての MQSeries Everyplace メッセージに関して、未確認メッセージ 1 つにつき 1 個のレコードがキューに含まれます。不安定なシステムでは、時間の経過とともに未確認メッセージのレコードが同期キューに蓄積されて、 MQSeries-ブリッジのパフォーマンスが低下します。

クライアント接続の***同期キュー除去間隔*** パラメーターが指定する時間間隔ごとに、クライアント接続に定義された同期キュー除去ルール・クラスが呼び出されて、古い未確認メッセージ・レコードを処理します。提供されたメッセージを削除してもよい場合、このルールはブール値 true を戻し、メッセージを残す必要がある場合は false を戻します。また、管理者はこのルールを使用して、たとえば一定時間後にメッセージ受信が確認されない場合に、アラートを発行して適切な処置を実行することもできます。

詳しくは、examples.mqbridge.rules.MQeSyncQueuePurgerRules を参照してください。

注: 同期キューを使用して MQSeries 伝送キュー・リスナーの状態メッセージを保管している場合、これらのメッセージはこのルールによって影響を受けません。

MQeStartupRule

ブリッジ、プロキシー、クライアント接続、またはリスナーのいずれかのオブジェクトがサーバー開始時にロードされるとき、管理対象の各オブジェクトに関してこのルールが参照されます。こうして、管理オブジェクトを開始するか、停止状態のままにするか、およびオブジェクトの子を同時に開始するかどうかが決定されます。

MQeStartupRule.permit(...) メソッドからの戻り値は、管理オブジェクトを開始するかどうかを示しています。可能な戻り値、およびそれぞれの効果は以下のとおりです。

START_NOTHING
この管理オブジェクトを開始しない。これは、管理オブジェクトに**「停止」**管理メッセージを送るのと同じ効果があります。

START_PARENTS_AND_ME
この管理オブジェクト、およびそのすべての親を開始する。これは、管理オブジェクトに対して、*affect-children* フラグ値 false とともに**「開始」**管理メッセージを送るのと同じ効果があります。

START_PARENTS_AND_ME_AND_CHILDREN
この管理オブジェクト、およびそのすべての親と子を開始する。これは、管理オブジェクトに対して、*affect-children* フラグ値 true とともに**「開始」**管理メッセージを送るのと同じ効果があります。

戻り値はアプリケーションによって制御することができるので、インテリジェントなルールをインプリメントすることができます。そのようなルールでは、たとえば、接続したい MQSeries システムがアクティブである場合に、MQSeries 伝送キューのみを開始することができます。

デフォルト構成ですべての管理オブジェクトに適用される com.ibm.mqe.mqbridge.MQeStartupRule は、 (ソース・コードが提供されている) クラス examples.mqbridge.rules.MQeStartupRule と類似しています。これら 2 つのクラスは、常に値 START_PARENTS_AND_ME を戻します。

各国語サポートの考慮事項

この節では、異なる言語を使用する MQSeries システム間を流れるメッセージを、MQSeries-ブリッジ がどのように処理するかを説明します。 217ページの図36 の図は、MQSeries Everyplace クライアント・アプリケーションから MQSeries アプリケーションへのメッセージのフローを示したものです。

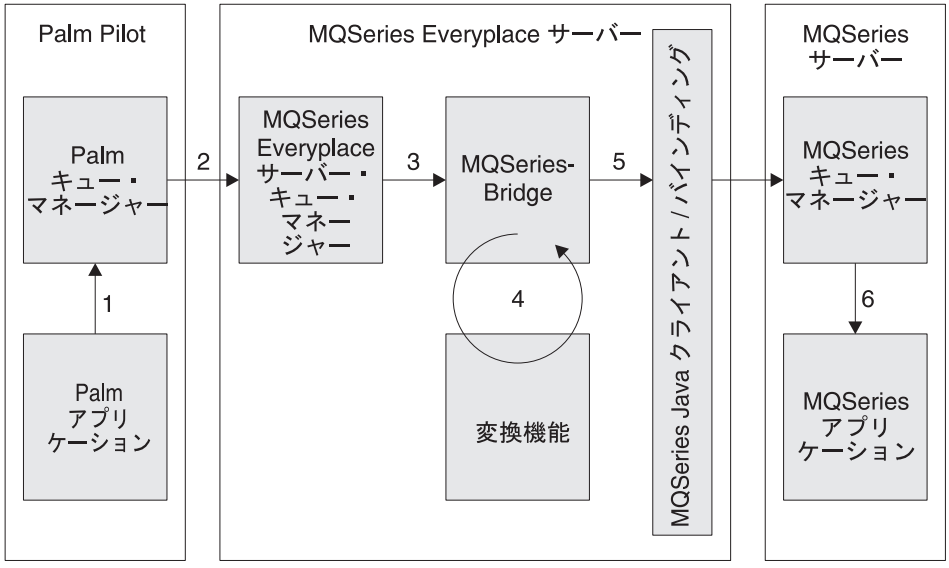

図 36. MQSeries Everyplace から MQSeries へのメッセージ・フロー

1. **クライアント・アプリケーション**
   a. クライアント・アプリケーションは、次のデータを含む MQSeries Everyplace メッセージ・オブジェクトを作成します。

      Unicode フィールド
      このストリングは、クライアント・マシン上で使用可能な該当するライブラリーを使用して生成されます (C/C++ を使用している場合)。

      バイト・フィールド
      このフィールドは変換してはなりません。

      ASCII フィールド
      このストリングには、ASCII 規格に準拠した、厳格に制限された有効文字が使用されます。有効文字は、すべての ASCII コード・ページ上で不変の文字だけです。

   b. メッセージはパーム・キュー・マネージャーに書き込まれます。この書き込み中に、変換は行われません。
2. **クライアント・キュー・マネージャーは、サーバー・キュー・マネージャーに書き込みを行います。**
   このステップ中に、メッセージは変換されません。
3. **MQSeries Everyplace サーバーは、メッセージを MQSeries-ブリッジ・キューに書き込みます。**
   このステップ中に、メッセージは変換されません。

4. **MQSeries-ブリッジは MQSeries Everyplace メッセージをユーザー作成の変換機能に渡します。**

   変換機能はまた、次のようにして MQSeries メッセージを作成します。

   - 以下を使用して、MQSeries Everyplace メッセージの中の Unicode フィールドが検索されます。

     ```
     String value = MQemsg.GetUnicode(fieldname)
     ```

   - 検索された値は、**MQmsg.writeChars(value)** を使用して MQSeries メッセージにコピーされます。
   - 以下を使用して、MQSeries Everyplace メッセージの中のバイト・フィールドが検索されます。

     ```
     Byte value = MQemsg.getByte(fieldName)
     ```

   - 検索された値は **MQmsg.writeByte(value)** を使用して MQSeries メッセージにコピーされます。
   - MQSeries 内に unicode 値を作成する場合、 MQSeries Everyplace メッセージ内の ascii フィールドは、 **MQmsg.writeChars(value)** を使用して検索され、コード・セット依存値を作成する場合は、 **MQmsg.writeString(value)** を使用して検索されます。

   writeString() を使用する場合は、ストリングの文字セットも使用することができます。変換機能は、その結果作成された MQSeries メッセージを、呼び出し元の MQSeries-ブリッジ・コードに戻します。

5. **MQSeries-ブリッジは、MQSeries java クラスを使用して、メッセージを MQSeries に渡します。**

   MQSeries メッセージ内の Unicode 値は、必要に応じて、ビッグ・エンディアンからリトル・エンディアンへ、またはその逆へ変換されます。 MQSeries メッセージ内のバイト値は、必要に応じて、ビッグ・エンディアンからリトル・エンディアンへ、またはその逆へ変換されます。 **writeString()** を使用して作成されたフィールドは、メッセージが MQSeries に書き込まれるときに、MQSeries java クラス内部の変換ルーチンを使用して変換されます。 ASCII データは、実行される文字セット変換にかかわりなく、ASCII データを残す必要があります。このステップ中に実行される変換は、メッセージのコード・ページ、送信 MQSeries java クライアント接続の CCSID、および受信 MQSeries サーバー接続チャネルの CCSID によって、異なる仕方で実行されます。

6. **メッセージは、MQSeries アプリケーションによって取得されます。**

   メッセージに Unicode ストリングが含まれる場合、アプリケーションはそのストリングを Unicode ストリングとして処理する必要があるか、またはそれを他の何らかのフォーマット (UTF8 など) に変換する必要があります。メッセージにバイト・ストリングが含まれる場合、アプリケーションはそのバイトをそのまま使用することができます (生データ)。メッセージにストリングが含まれる場合、それがメッセージから読み取られます。さらに、これが、アプリケーションで必要とされる異なるデー

タ・フォーマットに変換されることがあります。これは、 *characterSet* ヘッダー・フィールドのコード・セット値に応じて変換されます。 Java クラスはこの機能を自動的に提供します。

結論

MQSeries Everyplace アプリケーションを使用しており、文字関連データを MQSeries Everyplace から MQSeries へ伝達したい場合、どのメソッドを使用するかは、伝達したいデータによっておおかた決まります。

- **ASCII 文字コード・ページの可変範囲の文字がデータに含まれる場合** (各種の ASCII コード・ページ間で変更するときに、コード・ポイントの文字が変更されます)、**putUnicode** (コード・ページ間の変換の対象にはなりません)、または **putArrayOfByte** (送信側のコード・ページと受信側のコード・ページとの間の変換を処理しなければならない場合) のいずれかを使用することができます。

  注: ASCII コード・ページの可変部分の文字が変換の対象となるときには、**putAscii()** を***使用しないでください。***
- **データに ASCII 文字コード・ページの不変範囲の文字だけが含まれる場合には、** **putUnicode** (コード・ページ間の変換の対象にはなりません)、または **putAscii** (すべてのデータが ASCII コード・ページの不変範囲内にあるときには、コード・ページ間の変換の対象になりません) を使用することができます。

サンプル・ファイル

MQSeries-ブリッジ の機能をサポートする MQSeries Everyplace プログラムの作成および使用方法を示すサンプル・ファイルについては、 13ページの『第2章 概説』を参照してください。

第8章 セキュリティー

このセクションでは、 MQSeries Everyplace が提供するセキュリティー機能についての情報を説明します。さまざまなセキュリティーのレベルが、典型的な使用法のシナリオ、および使用法ガイダンスと共に説明されています。

セキュリティー機能

MQSeries Everyplace には、統合された一群のセキュリティー機能が備えられており、データがローカルに保管されている時も転送されている時にも、データを保護することができます。セキュリティーには、以下の 3 種類のカテゴリーがあります。

ローカル・セキュリティー
ローカル・セキュリティーは、いかなる MQSeries Everyplace データも保護します。

キュー・ベースのセキュリティー
キュー・ベースのセキュリティーは、開始キュー・マネージャーとキューの間、キュー上、およびキューと受信キュー・マネージャーの間の MQSeries Everyplace メッセージ・データを自動的に保護します。この保護は、宛先キューがローカルまたはリモート・キュー・マネージャーのどちらに所有されているかには関係ありません。

メッセージ・レベルのセキュリティー
メッセージ・レベルのセキュリティーでは、発信側と受信側の MQSeries Everyplace アプリケーションの間で、メッセージ・データを保護します。

キュー・ベースのセキュリティーは、MQSeries Everyplace によって内部的に処理されるので、メッセージの送信側および受信側で特別なアクションは必要ありません。ローカル・セキュリティーおよびメッセージ・レベルのセキュリティーは、アプリケーションによって起動する必要があります。

3 つのカテゴリーはすべて、属性 (MQeAttribute またはそれより下のもの) を適用することにより、メッセージ・データを保護します。属性は、カテゴリーに応じて、明示的か暗黙的に適用されます。

すべての属性には、次のオブジェクトの一部またはすべてが含まれます。
- 認証プログラム
- 暗号機能
- 圧縮機能
- キー
- 宛先エンティティー名

これらのオブジェクトが使用される方法は、 MQSeries Everyplace セキュリティーのカテゴリーに依存します。セキュリティーの各カテゴリーは、この章の後の方で詳しく説明されます。

MQSeries Everyplace は、次のサービスを提供して、セキュリティーの補助も行います。

私用レジストリー・サービス

MQSeries Everyplace 私用レジストリーは、公開および私用オブジェクトを保管できるリポジトリーや、私用レジストリーへのアクセスが、許可されたユーザーに制限されるように (ログイン) PIN 保護アクセスを提供します。また、機能がエンティティーの秘密鍵 (ディジタル署名、および RSA 暗号化解除用の) を PrivateRegistry インスタンスをそのままにして私用信任状がなくても使用できるように、付加的なサービスを提供します。

これらのサービスは、キュー・ベースのセキュリティー、および MQeTrustAttribute を使用するメッセージ・レベルのセキュリティーによって使用されます。

公開レジストリー・サービス

MQSeries Everyplace 公開レジストリーは、ミニ認証へ公的にアクセスできるリポジトリーを提供します。

これらのサービスは、キュー・ベースおよびメッセージ・レベルのセキュリティーによって使用されます。

ミニ認証発行サービス

MQSeries Everyplace には、デフォルトの*ミニ認証発行サービス* があります。これは、注意深く制御されるエンティティー名のセットに、ミニ認証を発行するように構成できます。

これらのサービスは、キュー・ベースおよびメッセージ・レベルのセキュリティーによって使用されます。

これらのサービスについては、この章の後の方で詳しく説明されます。

ローカル・セキュリティー

ローカル・セキュリティーは、メッセージ (MQeMsgObject) オブジェクトを含む、MQSeries Everyplace データ (MQeFields) オブジェクトを保護します。保護されたデータは、バイト配列で戻されます。データ・オブジェクトにローカル・セキュリティーを適用する場合には、以下のことを行わなければなりません。

1. 適切な認証プログラム、暗号化プログラム、および圧縮機能を使用して、属性を作成する
2. (パスワードまたはパスフレーズ・シードを入力して) 適切な キー をセットアップする

3. そのキーを明示的に属性に付加し、その属性をデータ (MQeFields) オブジェクトに付加して、データ・オブジェクトで **dump()** メソッドを起動する。

認証プログラムは、データへのアクセスが制御される方法を決定します。暗号化プログラムは、データの機密性を保護する暗号の強度を決定します。圧縮機能は、メッセージが要求するストレージを決定します。

MQSeries Everyplace には、ローカル・セキュリティーの使用を援助する MQeLocalSecure クラスが備えられています。ただし、適切な属性をセットアップして、パスワードまたはパスフレーズ・キーを提供することは、ローカル・セキュリティー・ユーザーの役割です。 MQeLocalSecure は、データを保護し、保管し、バックアップ・ストレージから復元する機能を備えています。アプリケーションが MQeLocalSecure を使用しないでメッセージに属性を付加する場合、**ダンプ**を使用した後にデータを保護し、**復元**を使用する前にデータを取り出すことも必要です。

使用法のシナリオ

さまざまな顧客サイトを扱う移動可能なエージェントが、ある顧客の機密データを偶発的に他の顧客と共用しないようにする場合のシナリオを考えてみましょう。ローカル・セキュリティー機能は、異なるキー、およびおそらく異なる暗号強度を使用して、単一のマシンで保留されている異なる顧客データを保護するため、単純なメソッドを提供します。

このシナリオを簡単に拡張すると、 MQSeries Everyplace サーバー・ノードにあるセキュア・キューから pull されるキーを使用して、保護されるローカル・データにアクセスします。エージェント・クライアントは、サーバー・キューにアクセスするためにそれ自体を認証し、ローカル・キー・データを pull する必要がありますが、実際のキーは分かりません。

このアプローチを行う利点の 1 つは、顧客の特定のデータすべてにアクセスするため、監査証跡を容易に集計できるということです。

セキュア機能の選択

MQeLocalSecure の使用時には、次の属性選択項目を選択できます。

認証プログラム
例 NTAuthenticator または UserIdAuthenticator

暗号機能
対象暗号機能 MQeDESCryptor、 MQe3DESCryptor、 MQeRC4Cryptor、 MQeRC6Cryptor または MQeMARSCryptor のいずれか

圧縮機能
MQeLZWCompressor、 MQeRleCompressor、または MQeGZIPCompressor

選択基準

未許可ユーザーによるローカル・データへのアクセスから保護するための付加的な制御を提供する必要が生じる場合、認証プログラムを使用します。キー・パスワードまたはパスフレーズを提供すると、この秘けつを知っているユーザーに対してアクセスが自動的に制限されるので、認証プログラムを使用する必要がなくなる場合もあります。

暗号機能は、必要な保護の強度によって決まります。暗号化が強ければ強いほど、アタッカーがデータへの不正なアクセスを試みるのは難しくなります。 128 ビットのキーを使用する対称暗号で保護されるデータは、もっと短いキーを使って暗号機能を使用する保護データより、アタックが難しくなります。しかし、暗号の強度に加えて、暗号機能の選択も他のさまざまな要因によって左右されます。例として、監査の承認を受けるため、トリプル DES を使用する必要がある金融業でのソリューションがあります。

保護データのサイズを最適化する必要がある場合には、圧縮機能を使用します。しかし、圧縮機能の効率性は、データの内容に依存しています。 MQeRleCompressor は長さのエンコードを実行します。つまり、圧縮機能のルーチンが、繰り返されるバイトを圧縮したり展開したりします。したがって、何度も繰り返されるバイトでデータを圧縮または圧縮解除する際に効率的です。 MQeLZWCompressor は LZW スキームを使用します。LZW アルゴリズムの最も単純な形式では、さまざまな語 (データ・パターン) が異なるコードに対して保管されている、ディクショナリー・データ構造を使用します。圧縮機能は、データに繰り返される語 (データ・パターン) が大量にある場合に最も効率的です。

使用法のガイド

1. 次のプログラム・フラグメントは、 MQeLocalSecure を使用して、 MQeFields オブジェクトを保護します。

```
try
   {
   .../*SIMPLE PROTECT FRAGMENT */
   .../*instantiate a DES cryptor */
    MQeDESCryptor desC =new MQeDESCryptor();
   .../*instantiate an Attribute using the DES cryptor */
    MQeAttribute attr =new MQeAttribute(null,desC,null);
   .../*instantiate a base Key object */
    MQeKey localkey =new MQeKey();
    .../*set the base Key object local key */
       localkey.setLocalKey("my secret key");
    .../*attach the key to the attribute */
    attr.setKey(localkey);
    /*instantiate a MQeFields object */
    MQeFields myData =new MQeFields();
    /*attach the attribute to the data object */
    myData.setAttribute(attr);
    /*add some test data */
    myData.putAscii("testdata","0123456789abcdef....");
    trace ("i:test data in ="+myData.getAscii("testdata"));
    /*encode the data */
```

```
    byte []protectedData =myData.dump();
    trace ("i:protected test data ="+MQe.byteToAscii(protectedData));
    }
catch (Exception e )
    {
    e.printStackTrace();/*show exception */
    }
try
    {
    .../* SIMPLE UNPROTECT FRAGMENT                                    */
    .../* instantiate a DES cryptor                                    */
    MQeDESCryptor des2C = new MQeDESCryptor( );
    .../* instantiate an attribute using the DES cryptor              */
    MQeAttribute des2A  = new MQeAttribute( null, des2C, null);
    .../* instantiate a (a helper) LocalSecure object                 */
    MQeLocalSecure ls2   = new MQeLocalSecure( );
    .../* open LocalSecure obj identifying target file and directory */
    ls2.open( ".¥¥", "TestSecureData.txt" );
    .../* use LocalSecure read to restore from target and decode data*/
    String outData       = MQe.byteToAscii( ls2.read( desA2,
                                          "It_is_a_secret" ) );
    .../* show results....                                           */
    trace ( "i: test data out = " + outData);
    ...
    }
  catch ( Exception e )
    {
    e.printStackTrace();               /* show exception          */
    }
```

2. 次のプログラム断片は、 MQeLocalSecure を使用しないで、ローカルに MQeMsgObject を保護します。

```
try
{
   .../*SIMPLE PROTECT FRAGMENT */
   .../*instantiate a DES cryptor */
 MQeDESCryptor desC = new MQeDESCryptor();
   .../*instantiate an Attribute using the DES cryptor */
 MQeAttribute attr = new MQeAttribute(null,desC,null);
   .../*instantiate a base Key object */
 MQeKey localkey = new MQeKey();
 .../*set the base Key object local key */
 localkey.setLocalKey("my secret key");
 .../*attach the key to the attribute */
 attr.setKey(localkey);
  /*instantiate an MQeFields object */
 MQeFields myData = new MQeFields();
 /*attach the attribute to the data object */
 myData.setAttribute(attr);
  /*add some test data */
 myData.putAscii("testdata", "0123456789abcdef....");
 trace ("i:test data in = " + myData.getAscii("testdata"));
  /*encode the data */
 byte [] protectedData = myData.dump();
 trace ("i:protected test data = " + MQe.byteToAscii(protectedData));
```

```
}
catch (Exception e )
{
     e.printStackTrace();  /*show exception */
}
try
{
 .../*SIMPLE UNPROTECT FRAGMENT */
 .../*instantiate a DES cryptor */
  MQeDESCryptor desC2 = new MQeDESCryptor();
 .../*instantiate an Attribute using the DES cryptor */
  MQeAttribute attr2 = new MQeAttribute(null,desC2,null);
 .../*instantiate a base Key object */
  MQeKey localkey2 = new MQeKey();
 .../*set the base Key object local key */
localkey2.setLocalKey("my secret key");
 .../*attach the key to the attribute */
attr2.setKey(localkey2 );
 /*instantiate a new data object */
MQeFields myData2 = new MQeFields();
 /*attach the attribute to the data object */
myData2.setAttribute(attr2 );
 /*decode the data */
myData2.restore(protectedData );
 /*show the unprotected test data */
trace ("i:test data out = " + myData2.getAscii("testdata"));
}
catch (Exception e )
{
   e.printStackTrace();   /*show exception */
   }
```

キュー・ベースのセキュリティー

キュー・ベースのセキュリティーは、開始キュー・マネージャーとキューの間、キュー上、およびキューと受信キュー・マネージャーの間の MQSeries Everyplace メッセージ・データを自動的に保護します。この保護形式は、宛先キューが属性で定義されている必要があります。この保護は、キューがローカルまたはリモート・キュー・マネージャーのどちらに属しているかには関係ありません。

このセキュリティーの単純な例は、NTAuthenticator、 MQe3DESCryptor、および MQeRleCompressor を持つ属性で定義される宛先キューです。そのような宛先キューが **putMessage**、 **getMessage**、または **browseMessages** を使用する宛先キューを使ってアクセスされると (ローカルでもリモートでも)、キュー属性が自動的に適用されます。この例では、アクセスを開始するアプリケーションは、操作が許可される前に NTAuthenticator の要件を満たす必要があります。操作が許可される場合、メッセージ・データは属性の MQe3DESCryptor および MQeRleCompressor を使用して、自動的にエンコードおよびデコードされます。例の宛先キューがリモートにアクセス (たとえば **putMessage** を使用して) されると、キュー・ベースのセキュリティーは、メッセージ・データがキュー属性の定義するレベルで自動的に保護されるようにします。 この保

護は、開始キュー・マネージャーとキューの間の転送中、メッセージがキューに保管されている間、およびキューと受信キュー・マネージャーの間の転送中に適用されます。

使用法のシナリオ

MQSeries Everyplace キュー・ベースのセキュリティーは、キュー・マネージャー間で転送されているメッセージ・データの機密性を保護する必要があるときに、いつでも使用できます。

典型的なシナリオとして、インターネットなどのオープン・ネットワーク経由で送達されるサービスがあります。この場合、開始アプリケーションがクライアント上のキュー・マネージャーを使用して要求を行い、サーバー・キュー・マネージャー・アプリケーションが提供するサービスにアクセスします。

これは次のように実現されます。

1. 開始クライアント・キュー・マネージャー・アプリケーションが、要求を MQSeries Everyplace メッセージにカプセル化する
2. **putMessage** を使用して、リモート・サーバーにある `XXX_service_request` と呼ばれる、キューにメッセージを転送する
3. サーバー上のキュー・マネージャー・アプリケーションが、`XXX_service_request` キューでメッセージを listen するようにセットアップされる
4. メッセージ・イベントが発生すると、**getMessage** が実行され、サービス要求メッセージを入手する
5. 要求が処理される (たとえば、バックエンド・システムで CICS トランザクションを起動することによって)
6. 応答 (トランザクション結果) がメッセージにカプセル化される
7. **putMessage** を使用して、開始クライアント・キュー・マネージャー上にある `XXX_service_reply` と呼ばれるキューに応答を戻す
8. 開始キュー・マネージャーで **waitForMessage** を使用して、応答メッセージが `XXX_service_reply` と呼ばれるローカル・キューに到達するのを待つ

この単純な例をサポートする 1 つの方法として、次のキューを定義します。

開始クライアント・キュー・マネージャーが所有する (たとえば ClientQMgr)

- TestClient_HomeServerQ
- XXX_service_reply

たくさんの選択項目がありますが、`TestClient_HomeServerQ` *`TimerInterval`* オプションをたとえば `5000` に設定すると、ポーリング間隔が 5 秒に設定され、クライアント・キュー・マネージャーがサーバー・キュー・マネージャーをポーリングするように起動します。このポーリングは、クライアント・キュー・マネージャーに送信されている、サーバー・キュー・マネージャーの store-and-forward についてのメッセージを「pull」します。また、クライアン

ト・キュー・マネージャー・アプリケーションを実行する前に、*AddQueueManager* オプションを使って、サーバー・キュー・マネージャーに参照を追加する必要があります。

サーバー・キュー・マネージャーが所有する (たとえば ServerQMgr)

- TestServer_StoreAndForwardQ
- XXX_service_request

このシナリオで使用するために `TestServer_StoreAndForwardQ` を定義するには、次の 2 つのステップが必要です。

1. キューを作成する
2. `ClientQMgr` という名前を使って、setAction MQeStoreAndForwardQueueAdminMsg.Action_AddQueueManager を実行する

セキュア機能の選択

キュー・ベースのセキュリティーを使用する際には、次の選択項目をすべて使用できます。

認証プログラム

NTAuthenticator または UserIdAuthenticator (または examples.attributes.LogonAuthenticator の下位にあるその他のもの)、あるいは MQeWTLSCertAuthenticator

暗号機能

MQeXORCryptor または対称暗号機能 MQeDESCryptor、MQe3DESCryptor、MQeRC4Cryptor、MQeRC6Cryptor、または MQeMARSCryptor のいずれか

圧縮機能

MQeLZWCompressor、MQeRleCompressor、または MQeGZIPCompressor

選択基準

キュー・ベースのセキュリティーは、同期キューを使用するために設計されたソリューションに適しています。この場合、選択基準は実際に (同期) キュー属性の認証プログラム、暗号機能、および圧縮機能の選択と関係しています。

認証プログラムを使用するオプションは、未許可ユーザーによるローカル・データへのアクセスから保護するための、付加的な制御を提供する必要が生じたために準備されています。これは、キュー・データがローカルにアクセスされる場合もリモートにアクセスされる場合も等しく当てはまります。

LogonAuthenticator の下位にあるもの (NTAuthenticator または UserIdAuthenticator) を使用すると、属性が活動化されるとき (たとえばアプリケーションがキュー上でデータの **putMessage()**、 **getMessage()** または **browseMessages()** を実行するとき)、操作が許可される前にオーセンティケーターの要件が満たされていなければなりません。キュー・ベースの 227ページの『使用法のシナリオ』では、 `XXX_service_request` キューが NTAuthenticator を含む属性で定義される場合、サーバー `XXX_service_request` キューへのアクセス (たとえば **putMessage()** 要求をクライアント・キュー・マネージャーからこのキューに送る場合) が、ターゲット・サーバーのドメインで有効な NT ユーザーとして定義される一連のユーザーに制限されます。 NTAuthenticator が例として提供され、より小さいユーザーのセットに対してより精密に細分化された制御を行える子孫を、簡単に作成できるようにします。

MQeWTLSCertAuthenticator を使用すると、この認証プログラムを使用する属性で保護されるキューへのすべてのリモート・アクセスが相互認証を完了してから、操作を実行することができます。交換されるミニ認証の相互認証は、受け取るミニ認証の妥当性検査をする各参加プログラムから構成されます。この妥当性検査は、受け取ったミニ認証が、要求側の独自のミニ認証と同じミニ認証サーバーによって署名した有効な署名付きエンティティーであること、および日付が有効であること (つまり現在日付が開始日付

より前であったり、終了日付より後であったりしないこと) を検査します。管理オプションによって、ソリューション作成者は、宛先キュー・マネージャーが独自の証明書 (独自のミニ認証および関連する秘密鍵を持つ独自の権限内の認証可能なエンティティー) を持つか、または所有するキュー・マネージャーの証明書を共用するかを選択することができます。キュー・ベースの
227ページの『使用法のシナリオ』では、`XXX_service_request` キューが MQeWTLSCertAuthenticator を含む属性で定義される場合、サーバー `XXX_service_request` キューへのアクセス (たとえば開始クライアント・キュー・マネージャー・アプリケーションがリモート **putMessage()** を実行する場合) は、開始クライアント・キュー・マネージャー、および正常に相互認証された宛先 `XXX_service_request` キューの信任状に依存します。

暗号機能は、必要な保護の強度、つまり、不法なアクセスを行おうとして保護データを暗号的にアタックするときに、アタッカーが直面する障害の程度に基づいて決定します。 128 ビットのキーを使用する対称暗号で保護されるデータは、もっと短いキーを使って暗号機能を使用する保護データより、アタックが難しくなります。しかし、暗号の強度に加えて、暗号機能の選択も他のさまざまな要因によって左右されます。この例として、監査の承認を受けるため、トリプル DES を使用する必要がある金融業でのソリューションがあります。

圧縮機能を使用するオプションは、保護データのサイズを最適化する必要に応じて決定されます。しかし、圧縮機能の効率性は、データの内容に依存しています。 MQeRleCompressor は長さのエンコードを実行します。つまり、圧縮機能のルーチンが、繰り返されるバイトを圧縮したり展開したりします。したがって、何度も繰り返されるバイトでデータを圧縮または圧縮解除する際に効率的です。 MQeLZWCompressor は LZW スキームを使用します。LZW アルゴリズムの最も単純な形式では、さまざまな語 (データ・パターン) が異なるコードに対して保管されている、ディクショナリー・データ構造を使用します。圧縮機能は、データに繰り返される語 (データ・パターン) が大量にある場合に最も効率的です。

使用法のガイド

キュー・ベースのセキュリティーを使用するには、キューを所有しているキュー・マネージャーが私用レジストリーを持っている必要があります。 MQeWTLSCertAuthenticator が使用される場合には、レジストリーも独自の信任状を持っている必要があります。これは、ミニ認証サーバー による自動登録によって獲得します。以下の例では、キュー・マネージャーの構成 (.ini) ファイルの Registry セクションに情報を追加することによって、信任状処理を行えるようにします。 MQeWTLSCertAuthenticator を使用しない場合にも私用レジストリーは必要ですが、信任状を入手するためにミニ認証サーバーに登録する必要はありません。

次のコード・フラグメントは、キュー・マネージャー・インスタンスを作成し、
227ページの『使用法のシナリオ』で説明されているキュー・ベースのシナリオについて識別されるキューを定義する方法の例を示します。アプリケーションを開始するクライ

アント・キュー・マネージャー、およびサーバー・キュー・マネージャー AppRunList で開始されるアプリケーションのフラグメントも示されています。

SimpleCreateQM を使用して ClientQMgr および ServerQMgr インスタンスを作成する

注: このプログラム例では、*PIN*、*認証要求 PIN* および *鍵リング・パスワード* が構成ファイルに保管されている必要があります。これは、例を示す際の都合によるもので、実動システムではお勧めしません。 PIN およびパスワードが、許可を受けずに開示されないように注意してください。

SimpleCreateQM は、ユーザーによる、私用レジストリーを持つキュー・マネージャー・インスタンスの作成を支援します。クラスは、MQePrivateClient1.ini および MQePrivateServer1.ini の Registry セクションにあるパラメーターを使用します。

特定のインスタンスは、次のように作成されます。

1. MQePrivateClient1.ini および MQePrivateServer1.ini の Registry セクションにある私用レジストリー関連パラメーターを、デフォルトから必要な設定にリセットします。

   ```
   (ascii)LocalRegType=PrivateRegistry
   (ascii)DirName=.¥¥MQeNode_PrivateRegistry
   (ascii)PIN=12345678
     < change PIN from '12345678' to the PIN to be provided subsequently at
        queue manager  start-up time to enable the queue manager to access its
       own private registry >
   ```

   MQeWTLSCertAuthenticator が使用される場合のみ、次の 3 つのキーワード (*CertReqPIN*、*KeyRingPassword* および *CAIPAddrPort*) を含めてください。

   ```
   (ascii)CertReqPIN=12345678
     < change CertReqPIN from '12345678' to a new value that matches the value set by
       Mini Certificate Server's Administrator when the queuemanager instance is defined >
   (ascii)KeyRingPassword=It_is_a_secret
     < change the KeyRingPassword  from 'It_is_a_secret'  to the password that
        to be subsequently provided at queuemanager start-up time to enable
        the queuemanager instance to access its protected private credentials
        within its Private Registry. >
   (ascii)CAIPAddrPort=9.20.X.YYY:8082
     < change this to the IP address and port of the solution's
         MiniCertificateServer.>
   ```

2. 最後の 3 つのキーワードが提供されると、自動登録が起動されるので、キュー・マネージャー・インスタンスを追加する前に、MiniCerificateServerGUI を開始する必要があります。さらに「管理 (Administration)」モードを使って、キュー・マネージャー・インスタンス (`ClientQMgr` および `ServerQMgr`) を有効な認証可能エンティティーとして定義し、認証要求 PIN を前のステップの MQePrivateClient1.ini および MQePrivateServer1.ini ファイルの Registry セクションの `CertReqPIN=` 行で定義されたのと同じ値に設定することが必要です。
3. MiniCertificateServerGUI インスタンスを開始し、「サーバー (Server)」モードを選択します。

4. TestCreate プログラム (次のコード断片に示される) を実行して、キュー・マネージャー・インスタンスを作成します。

```
package test;
import com.ibm.mqe.*;
import examples.install.*;
public class TestCreate extends MQe
  {
  public void createQMs( )
    {
    /* start trace...                                                    */
    try{
       MQeTraceInterface trace =
        (MQeTraceInterface) MQe.loader.loadObject(
                                        "examples.awt.AwtMQeTrace" );
       trace.activate( "TestCreate...", null );
       }
    catch(Exception e) {e.printStackTrace(); }
    try{
       String INI_FileName   = ".¥MQePrivateClient1.ini";
       String QueueDir       = ".¥ClientQMgr¥Queues¥";
       SimpleCreateQM c_QMgr = new SimpleCreateQM();
       if ( c_QMgr.createQMgr(INI_FileName, QueueDir) )
         trace ( ">>>> ClientQMgr created OK...");
       else
         trace (">>>> error creating ClientQMgr...");
       INI_FileName          = ".¥MQePrivateServer1.ini";
       QueueDir              = ".¥ServerQMgr¥Queues¥";
       SimpleCreateQM s_QMgr = new SimpleCreateQM();
       if ( s_QMgr.createQMgr(INI_FileName, QueueDir) )
         trace ( ">>>> ServerQMgr created OK...");
       else
         trace (">>>> error creating ServerQMgr...");
       }
    catch (Exception e)
       {
       trace (">>>> SimpleCreateQM eception = "+ e.getMessage( ) );
       e.printStackTrace();
       }
    }
  public static void main(String args[])
    {
    TestCreate testc = new TestCreate( );
    testc.createQMs( );
    }
  }
```

前述のキュー・ベースのシナリオ用に識別されるキューの定義

キュー定義をキュー・マネージャー・インスタンスに追加するには、いくつかの方法があります。ここで説明されるメソッドは、キュー・マネージャー・インスタンスをローカルに開始し、関係のある管理メッセージを作成してキュー・マネージャーの独自の管理キューに送信することによって新しいキュー定義を追加し、その後 AdminReply キューで成功の確認を待機します。

ClientQMgr キュー - TestClient_HomeServerQ の追加:

MQePrivateClient クラスを使用して ClientQMgr をローカルに開始し、異なるバージョン MQePrivateClient2.ini (これは、 *PIN*、*KeyRingPassword*、および *CertReqPIN* のハードコーディング値を意図的に保持しない) を使用して管理メッセージを作成し、さらにそれを使ってキューを追加して、ポーリング時間間隔を設定します。

```
{
try{
   /* start ClientQMgr...                                              */
   String QMgrName       = "ClientQMgr";
   String QName          = "TestClient_HomeServerQ"
   MQeAttribute qattr    = new MQeAttribute(null,
                                      new MQe3DESCryptor, null);
   String FileDesc       = "MsgLog:.";
   MQePrivateClient newC = new MQePrivateClient(
                            ".//MQePrivateClient2.ini",
                            "12345678",         /* or new PIN       */
                            "It_is_a_secret",   /* or new KeyRingPwd*/
                             null);
   MQeQueueManager newQM =  newC.queueManager;
   /* create and use Admin msg to add HomeServerQ...                   */
   MQeHomeServerQueueAdminMsg msg  =
     new MQeHomeServerQueueAdminMsg("ServerQMgr",
                                    "ServerTestQ_StoreAndForward");
   MQeFields parms       = new MQeFields( );
   parms.putLong( MQeHomeServerQueueAdmin.Queue_QTmerInterval, 5000 );
   msg.setTargetQMgr( QMgrName );
   msg.setName( QMgrName, QName );
   msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
   msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
   msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
   msg.putArrayOfByte( MQe.Msg_CorrelID,
            Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
   MQeFields msgTest = new MQeFields( );
   msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                           msg.getArrayOfByte( MQe.Msg_CorrelID ) );
   parms.putAscii( msg.Queue_QMgrName, "ServerQMgr" );
   parms.putAscii( msg.Queue_FileDesc, FileDesc );

   if ( qattr.getAuthenticator( ) != null ) /*add qattr auth details*/
     {
     parms.putAscii( msg.Queue_Authenticator,
                                qattr.getAuthenticator( ).type( ) );
     if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
       {
       parms.putAscii( msg.Queue_AttrRule,
                                  "examples.rules.AttributeRule" );
       parms.putByte( msg.Queue_TargetRegistry,
                                       msg.Queue_RegistryQueue );
       }
     }
   if ( qattr.getCryptor( ) != null )
     {
```

```
      parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
      if (! parms.contains( msg.Queue_AttrRule ) )
        parms.putAscii( msg.Queue_AttrRule,
                                    "examples.rules.AttributeRule" );
      }
    if ( qattr.getCompressor( ) != null )
      parms.putAscii( msg.Queue_Compressor,
                               qattr.getCompressor( ).type( ) );
    parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
    msg.create( parms );
    trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
    /* use Admin msg to add HomeServerQ...                              */
    newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
    MQeAdminMsg respMsg      = null;
    trace(">>> Waiting for a response to create Admin Msg...");
    respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                               "AdminReplyQ", msgTest, null, 0, 3000);
    trace(">>> Admin Msg processed OK...");
    /* process Admin msg response                 ...                   */
    if ( respMsg == null )
      trace ( "i: create Queue failed, no response message received" );
    else
      {
      if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
        trace( "i: create Queue added queue OK..." );
      else
        trace( "i: create Queue failed: " + respMsg.getReason( ) );
      }
    newQM.close();
    }
  catch ( Exception e )
    {
    trace (">>>> add HomeServerQ exception = "+ e.getMessage( ) );
    e.printStackTrace();
    }
 }
```

ClientQMgr キュー - XXX_service_reply キューの追加:

MQePrivateClient クラスを使用して ClientQMgr をローカルに開始し、異なるバージョン MQePrivateClient2.ini (これは、 *PIN*、*KeyRingPassword* および *CertReqPIN* のハードコーディング値を意図的に保持しない) を使用して管理メッセージを作成し、さらにそれを使ってキューを追加します。

```
{
  try{
    /* start ClientQMgr...                                              */
    String QMgrName       = "ClientQMgr";
    String QName          = "XXX_service_reply"
    MQeAttribute qattr    = new MQeAttribute(null,
                                        new MQe3DESCryptor, null);
    String FileDesc       = "MsgLog:.";
    MQePrivateClient newC = new MQePrivateClient(
                             ".//MQePrivateClient2.ini",
```

```
                        "12345678",         /* or new PIN       */
                        "It_is_a_secret",   /* or new KeyRingPwd*/
                         null);
MQeQueueManager newQM =  newC.queueManager;
/* create and use Admin msg to add XXX_service_reply queue       */
MQeQueueAdminMsg msg  = new MQeQueueAdminMsg( );
MQeFields parms       = new MQeFields( );
msg.setTargetQMgr( QMgrName );
msg.setName( QMgrName, QName );
msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
msg.putArrayOfByte( MQe.Msg_CorrelID,
         Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
MQeFields msgTest = new MQeFields( );
msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                       msg.getArrayOfByte( MQe.Msg_CorrelID ) );
parms.putAscii( msg.Queue_QMgrName, "ServerQMgr" );
parms.putAscii( msg.Queue_FileDesc, FileDesc );
if ( qattr.getAuthenticator( ) != null )
  {
  parms.putAscii( msg.Queue_Authenticator,
                              qattr.getAuthenticator( ).type( ) );
  if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
    {
    parms.putAscii( msg.Queue_AttrRule,
                                "examples.rules.AttributeRule" );
    parms.putByte( msg.Queue_TargetRegistry,
                                       msg.Queue_RegistryQueue );
    }
  }
if ( qattr.getCryptor( ) != null )
  {
  parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
  if (! parms.contains( msg.Queue_AttrRule ) )
    parms.putAscii( msg.Queue_AttrRule,
                                "examples.rules.AttributeRule" );
  }
if ( qattr.getCompressor( ) != null )
  parms.putAscii( msg.Queue_Compressor,
                           qattr.getCompressor( ).type( ) );
parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
msg.create( parms );
trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
/* use Admin msg to add queue     ...                            */
newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
MQeAdminMsg respMsg      = null;
trace(">>> Waiting for a response to create Admin Msg...");
respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                           "AdminReplyQ", msgTest, null, 0, 3000);
trace(">>> Admin Msg processed OK...");
/* process Admin msg response                 ...                */
if ( respMsg == null )
  trace ( "i: create Queue failed, no response message received" );
else
```

```
      {
      if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
        trace( "i: create Queue added queue OK..." );
      else
        trace( "i: create Queue failed: " + respMsg.getReason( ) );
      }
    newQM.close();
    }
  catch ( Exception e )
    {
    trace (" >>> add XXX_service_reply Q excep = "+ e.getMessage( ) );
    e.printStackTrace();
    }
 }
```

ServerQMgr キュー - TestServer_StoreAndForwardQ の追加: MQePrivateClient クラスを使用して ServerQMgr をローカルに開始し、異なるバージョン MQePrivateServer2.ini (これは、 *PIN*、*KeyRingPassword* および *CertReqPIN* のハードコーディング値を意図的に保持しない) を使用して管理メッセージを作成し、さらにそれを使ってキューを追加してから、リモート・キュー・マネージャー参照を追加します。

```
{
  try{
    /* start ServerQMgr, locally                                    */
    String QMgrName       = "ServerQMgr";
    String QName          = "TestServer_StoreAndForwardQ"
    MQeAttribute qattr    = new MQeAttribute(null,
                                        new MQe3DESCryptor, null);
    String FileDesc       = "MsgLog:.";
    MQePrivateClient newC = new MQePrivateClient(
                             ".//MQePrivateServer2.ini",
                             "12345678",         /* or new PIN       */
                             "It_is_a_secret",   /* or new KeyRingPwd*/
                              null);
    MQeQueueManager newQM =  newC.queueManager;
    /* create and use Admin msg to add StoreAndForwardQ             */
    MQeStoreAndForwardQueueAdminMsg( ) msg  =
                                 new MQeStoreAndForwardQueueAdminMsg( );
    MQeFields parms       = new MQeFields( );
    msg.setTargetQMgr( QMgrName );
    msg.setName( QMgrName, QName );
    msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
    msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
    msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
    msg.putArrayOfByte( MQe.Msg_CorrelID,
            Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
    MQeFields msgTest = new MQeFields( );
    msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                          msg.getArrayOfByte( MQe.Msg_CorrelID ) );
    parms.putAscii( msg.Queue_QMgrName, QMgrName );
    parms.putAscii( msg.Queue_FileDesc, FileDesc );
    if ( qattr.getAuthenticator( ) != null )
      {
      parms.putAscii( msg.Queue_Authenticator,
```

```
                                qattr.getAuthenticator( ).type( ) );
  if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
    {
    parms.putAscii( msg.Queue_AttrRule,
                                  "examples.rules.AttributeRule" );
    parms.putByte( msg.Queue_TargetRegistry,
                                       msg.Queue_RegistryQueue );
    }
  }
if ( qattr.getCryptor( ) != null )
  {
  parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
  if (! parms.contains( msg.Queue_AttrRule ) )
    parms.putAscii( msg.Queue_AttrRule,
                                  "examples.rules.AttributeRule" );
  }
if ( qattr.getCompressor( ) != null )
  parms.putAscii( msg.Queue_Compressor,
                           qattr.getCompressor( ).type( ) );
parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
msg.create( parms );
trace(" >>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
/* use Admin msg to add queue     ...                              */
newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
MQeAdminMsg respMsg      = null;
trace(" >>> Waiting for a response to create Admin Msg...");
respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                           "AdminReplyQ", msgTest, null, 0, 3000);
trace(" >>> Admin Msg processed OK...");
/* process Admin msg response                  ...                  */
if ( respMsg == null )
  trace ( "i: create Queue failed, no response message received" );
else
  {
  if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
    trace( "i: create Queue added queue OK..." );
  else
    trace( "i: create Queue failed: " + respMsg.getReason( ) );
  }
/* use Admin msg to StoreAndForwardQ AddQueueManager reference  */
msg = new MQeStoreAndForwardQueueAdminMsg( );
msg.addQueueManager( "ClientQMgr" );
parms = new MQeFields( );
msg.setTargetQMgr( QMgrName );
msg.setName( QMgrName, QName );
msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
msg.putArrayOfByte( MQe.Msg_CorrelID,
         Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
MQeFields msgTest = new MQeFields( );
msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                       msg.getArrayOfByte( MQe.Msg_CorrelID ) );
parms.putAscii( msg.Queue_QMgrName, QMgrName );
parms.putAscii( msg.Queue_FileDesc, FileDesc );
```

```
    msg.setAction(
            MQeStoreAndForwardQueueAdminMsg.Action_AddQueueManager );
    trace(" >>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
    newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
    MQeAdminMsg respMsg      = null;
    trace(" >>> Waiting for a response to update Admin Msg...");
    respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                              "AdminReplyQ", msgTest, null, 0, 3000);
    trace(" >>> Admin Msg processed OK...");
    /* process Admin msg response                 ...                  */
    if ( respMsg == null )
      trace ( "i: create Queue failed, no response message received" );
    else
      {
      if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
        trace( "i: create Queue added queue OK..." );
      else
        trace( "i: create Queue failed: " + respMsg.getReason( ) );
      }
    trace(" >>> StoreAndForwardQ AddQueueManager reference OK..." );
    newQM.close();
    }
  catch ( Exception e )
    {
    trace (" >>> add StoreAndForwardQ exception = "+ e.getMessage( ) );
    e.printStackTrace();
    }
```

ServerQMgr キュー - XXX_service_request キューの追加:

MQePrivateClient クラスを使用して ServerQMgr をローカルに開始し、異なるバージョン MQePrivateServer2.ini (これは、 *PIN*、*KeyRingPassword* および *CertReqPIN* のハードコーディング値を意図的に保持しない) を使用して管理メッセージを作成し、さらにそれを使ってキューを追加します。

```
{
  try{
    /* start ServerQMgr...                                            */
    String QMgrName       = "ServerQMgr";
    String QName          = "XXX_service_request"
    MQeAttribute qattr    = new MQeAttribute(null,
                                    new MQe3DESCryptor, null);
    String FileDesc       = "MsgLog:.";
    MQePrivateClient newC = new MQePrivateClient(
                             ".//MQePrivateServer2.ini",
                             "12345678",         /* or new PIN       */
                             "It_is_a_secret",   /* or new KeyRingPwd*/
                              null);
    MQeQueueManager newQM =  newC.queueManager;
    /* create and use Admin msg to add XXX_service_request queue     */
    MQeQueueAdminMsg msg  = new MQeQueueAdminMsg( );
    MQeFields parms       = new MQeFields( );
    msg.setTargetQMgr( QMgrName );
    msg.setName( QMgrName, QName );
```

```
msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
msg.putArrayOfByte( MQe.Msg_CorrelID,
       Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
MQeFields msgTest = new MQeFields( );
msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                        msg.getArrayOfByte( MQe.Msg_CorrelID ) );
parms.putAscii( msg.Queue_QMgrName, QMgrName );
parms.putAscii( msg.Queue_FileDesc, FileDesc );

if ( qattr.getAuthenticator( ) != null ) /*add qattr auth details*/
  {
  parms.putAscii( msg.Queue_Authenticator,
                                qattr.getAuthenticator( ).type( ) );
  if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
    {
    parms.putAscii( msg.Queue_AttrRule,
                                  "examples.rules.AttributeRule" );
    parms.putByte( msg.Queue_TargetRegistry,
                                       msg.Queue_RegistryQueue );
    }
  }
if ( qattr.getCryptor( ) != null )
  {
  parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
  if (! parms.contains( msg.Queue_AttrRule ) )
    parms.putAscii( msg.Queue_AttrRule,
                                  "examples.rules.AttributeRule" );
  }
if ( qattr.getCompressor( ) != null )
  parms.putAscii( msg.Queue_Compressor,
                             qattr.getCompressor( ).type( ) );
parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
msg.create( parms );
trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
/* use Admin msg to add XXX_service_request queue                */
newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
MQeAdminMsg respMsg      = null;
trace(">>> Waiting for a response to create Admin Msg...");
respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                             "AdminReplyQ", msgTest, null, 0, 3000);
trace(">>> Admin Msg processed OK...");
/* process Admin msg response                 ...                 */
if ( respMsg == null )
  trace ( "i: create Queue failed, no response message received" );
else
  {
  if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
    trace( "i: create Queue added queue OK..." );
  else
    trace( "i: create Queue failed: " + respMsg.getReason( ) );
  }
newQM.close();
}
```

```
    catch ( Exception e )
       {
       trace (" >>> add XXX_service_request excep = "+ e.getMessage( ) );
       e.printStackTrace();
       }
     }
```

サーバー・キュー・マネージャー AppRunList が開始するアプリケーション:

このセクションでは、MQePrivateServer2.ini の拡張例を使用して、 ServerQMgr の始動時に自動的に開始される AppRunList アプリケーションの追加方法を示します。また、TestService アプリケーションの例も示します。

MQePrivateServer2.ini の例

```
MQePrivateServer2.ini - with AppRunList extension...
[Alias]
(ascii)EventLog=examples.log.LogToDiskFile
(ascii)Network=com.ibm.mqe.adapters.MqeTcpipHttpAdapter
(ascii)QueueManager=com.ibm.mqe.MqeQueueManager
(ascii)Trace=examples.awt.AwtMQeTrace
(ascii)MsgLog=com.ibm.mqe.adapters.MqeDiskFieldsAdapter
(ascii)FileRegistry=com.ibm.mqe.registry.MqeFileSession
(ascii)PrivateRegistry=com.ibm.mqe.registry.MqePrivateSession
(ascii)ChannelAttrRules=examples.rules.AttributeRule
(ascii)AttributeKey_1=com.ibm.mqe.MQeKey
(ascii)AttributeKey_2=com.ibm.mqe.attributes.MqeSharedKey
[ChannelManager]
(int)MaxChannels=0
[Listener]
(ascii)Listen=Network::8082
(ascii)Network=Network:
(int)TimeInterval=300
[QueueManager]
(ascii)Name=ServerQMgr
(ascii)QueueStore=MsgLog:.¥MQeNode_PrivateRegistry
[Registry]
(ascii)LocalRegType=PrivateRegistry
(ascii)DirName=.¥¥MQeNode_PrivateRegistry
(ascii)PIN=not set
(ascii)CertReqPIN=not set
(ascii)KeyRingPassword=not set
(ascii)CAIPAddrPort=9.20.X.YYY:8082
[AppRunList]
(ascii)App1=test.TestService
```

サーバー TestSevice アプリケーションの例

```
package test;
import com.ibm.mqe.*;
import com.ibm.mqe.attributes.*;
import java.util.*;
public class TestService extends MQe
   implements MQeRunListInterface, MQeMessageListenerInterface, Runnable
```

```
{
protected Thread applicationThread = null;
protected MQeQueueManager thisQMgr = null;

/* constructor                                                          */
public TestService( ) throws Exception
  {
  }

/* activate method                                                      */
public Object activate( Object    owner,
                        Hashtable loadTable,
                        MQeFields setupData ) throws Exception
  {
  System.out.println(" TestService, activate, owner objref = " + owner);
  thisQMgr           = (MQeQueueManager)owner;  /* save QMgr objref   */
  applicationThread  = new Thread(
               this, "applicationThread" ); /* create svr app thread */
  System.out.println(" TestService, activate no of active threads = " +
                                                 Thread.activeCount( ) );
  Thread t[] = new Thread[Thread.activeCount( )];
  int i = Thread.enumerate( t );
  for ( int j = 0; j < i; j++ )              /* look at svr threads   */
    System.out.println("TestService activate, active thread name = "
                                                      + t[j].getName( ) );
  applicationThread.start( );                /* start appl'n Thread.  */
  return this;
  }

/* run method                                                           */
public void run( )
  {
  System.out.println("TestService, Run...");
  /* add listener for XXX_service_request queue                        */
 try {
      thisQMgr.addMessageListener( this, "XXX_service_request",
                                                    new MQeFields( ) );
      }
  catch( Exception e)
      {
      e.printStackTrace( );
      }
  }

/* MessageArrived event handler                                         */
/* MsgArrived event is generated when a message arrives on a queue      */
public void messageArrived( MQeMessageEvent msgEvent )
{
 try {
    System.out.println(" TestService, msgEvent, messageArrived ");
    System.out.println(" TestService, msgEvent getQueueManagerName = " +
                                     msgEvent.getQueueManagerName() );
    System.out.println(" TestService, msgEvent getQueueName = " +
                                           msgEvent.getQueueName( ) );
    /* get XXX service request message                                  */
```

```
    MQeMsgObject reqmsg = thisQMgr.getMessage(
                          msgEvent.getQueueManagerName( ),
                          msgEvent.getQueueName( ),
                          msgEvent.getMsgFields( ),
                          null,
                          0);
    /* process service request here                                    */
    String reqdata   = reqmsg.getAscii("XXX_service_request_data");
    String replydata = reqdata + "_reply";
    /* build XXX_service reply message here                            */
    MQeMsgObject replymsg = new MQeMsgObject( );
    replymsg. putArrayOfByte( MQe.Msg_CorrelID,
                           reqmsg.getArrayOfByte(MQe.Msg_CorrelID ) );
    replymsg.putAscii("XXX_service_reply_data", replydata );
    System.out.println(" TestService, msgEvent putting service reply " +
                       "to ClientQMgr XXX_service_reply queue");
    /* put reply to ClientQMgr XXX_service reply queue                 */
    thisQMgr.putMessage( "ClientQMgr", "XXX_service_reply",
                                                 replymsg, null, 1 );
    }
catch( Exception e )
    {
    e.printStackTrace( );
    }
}
/* finalize method                                                    */
protected void finalize()
  {
  System.out.println("TestService, finalize...");
  applicationThread.stop( );
  applicationThread.destroy( );
  }
```

XXX_service_request を開始するクライアント・キュー・マネージャー・アプリケーション:

227ページの『使用法のシナリオ』のキュー・ベースのセキュリティーのシナリオ例は、MQeMsgObject の要求をカプセル化し、**putMessage()** を使用することによって XXX_service_request メッセージを開始し、サーバー・キュー・マネージャーの XXX_service_request キューに要求を確実に送達する、クライアント・キュー・マネージャー・アプリケーションについて説明します。その後、独自の XXX_service_reply キューで **waitForReply()** を使って、サービス要求への応答を待ちます。

シナリオでは、サーバー上の TestService アプリケーションは、 **getMessage()** を使って XXX_service_request キューからサービス要求を入手することによりサービス要求を処理し、(たとえばバックエンド・トランザクションの起動によって) 要求を処理してから、応答 MQeMsgObject を構築します。さらにサーバー・キュー・マネージャー **putMessage()** を使用して (リモート) 開始クライアント・キュー・マネージャーに応答を戻します。

サーバー・キュー・マネージャーは、メッセージを内部的に TestServer_StoreAndForwardQ に書き込みます。クライアント・キュー・マネージャーは、TestServer_StoreAndForwardQ からメッセージを pull し、それをその ClientTest_HomeServerQ で受け取ってから、意図した宛先 XXX_service_reply キューに書き込みます。

以下のクライアント・アプリケーションは、サービス要求の呼び出しと、結果の応答を処理する簡単な例を示しています。

```
package test;
import com.ibm.mqe.*;
import examples.queuemanager.*;
 public class UseTestService extends MQe
  {
  protected MQeQueueManager thisQMgr = null;
  /* serviceRequest method                                                  */
  public void serviceRequest( )
    {
    /* start trace...                                                        */
    try{
       MQeTraceInterface trace =
        (MQeTraceInterface) MQe.loader.loadObject(
                                        "examples.awt.AwtMQeTrace" );
       trace.activate( "UseTestService...", null );
       }
    catch(Exception e) {e.printStackTrace();
    /* start and use Client queuemanager to put request & process reply */
    try {
        /* start Client queue manager                                     */
       MQePrivateClient newC = new MQePrivateClient(
                                     ".//MQePrivateClient2.ini",
                                     "12345678",
                                     "It_is_a_secret",
                                     null );
       MQeQueueManager newQM =  newC.queueManager;
        /* build svc request and use putMessage to put it to server     */
        MQeMsgObject msgreq   = new MQeMsgObject( );
        long thisReq_CorrelID = newQM.uniqueValue();
        msgreq.putArrayOfByte( MQe.Msg_CorrelID,
                                      longToByte( thisReq_CorrelID) );
        String reqdata        = "0123456789abcdef";
        msgreq.putArrayOfByte("XXX_service_request_data",
                                           asciiToByte(reqdata) );
        newQM.putMessage("ServerQMgr","XXX_service_reqest",msgreq,null,1);
        trace( " >>> request put to ClientQMgr,XXX_service_request q OK");
        /* field and process reply to service request                    */
        trace( " >>> waiting for reply message...");
        MQeFields msgreq_filter = new MQeFields();
        msgreq_filter.putArrayOfByte( MQe.Msg_CorrelID,
                                      longToByte( thisReq_CorrelID) );
        MQeMsgObject msgreply  = newQM.waitForMessage( newQM.getName( ),
                     "XXX_service_reply", msgreq_filter, null, 0, 3000 );
        trace(" >>> service request reply = " +
          byteToAscii(msgreply.getArrayOfByte("XXX_service_reply_data")));
```

```
        }
    catch(Exception e2) { e2.printStackTrace();}
    }
  }
  public static void main(String args[])
    {
    UseTestService testsvc = new UseTestService( );
    testsvc.serviceRequest();
    }
  }
```

キュー・ベースのセキュリティーと自動登録のトリガー

キュー・マネージャーがリモート・キューまたは MQeWTLSCertAuthenticator を含む属性で定義されたローカル・キューにアクセスする場合、キュー・マネージャーおよびキューは認証可能なエンティティーであり、独自の信任状が必要です。

キュー・マネージャーの信任状は、自動登録を起動することによって作成されます。自動登録の起動の最も簡単な方法は、関係のあるキーワードを、キュー・マネージャーの作成時に使用される ini ファイルの Registry セクションに含めるという方法です。 ini ファイルの Registry セクションで必要なキーワードは、次のとおりです。

```
(ascii)CertReqPIN=12345678
  < change CertReqPIN '12345678' to a new value that matches the value set by
   Mini Certificate Server's Administrator when the Queue Manager instance is defined >
(ascii)KeyRingPassword=It_is_a_secret
  < change the default KeyRingPassword  from 'It_is_a_secret'  to the password that
     is to be subsequently provided at Queue Manager start-up time to enable
     the Queue Manager instance to access its protected private credentials
     within its Private Registry. >
(ascii)CAIPAddrPort=9.20.X.YYY:8082
  < change this to the IP address and port of the solution's MiniCertificateServer.>
```

キューの信任状 (MQeWTLSCertAuthenticator を含む属性を持つ) も、自動登録のトリガーによって作成されます。これは、キューを追加する管理メッセージが次の場合に処理される際に、自動的に発生します。

- 所有するキュー・マネージャーがすでに自動登録済みであり、独自の証明書とソリューションのミニ認証サーバーにアクセスするのに必要なパラメーターを使用して開始している
- 所有するキュー・マネージャー名とキュー名がミニ認証サーバー・アドミニストレーターによって定義済みであり、ミニ認証要求 PIN が、所有するキュー・マネージャーを開始するのに使用される *CertReqPIN* 値と同じ値に設定されている
- ミニ認証サーバーが使用可能、開始済み、および「サーバー (Server)」モードである

(MQeWTLSCertAuthenticator を含む属性を持つ) キューを追加すると、キューは独自の信任状を持つか、または所有するキュー・マネージャーの信任状を共用できます。どちらになるかは、キュー作成管理メッセージの構成時に決定されます。次のコード断片は、関係のあるパラメーターとその意味を示します。

ServerQMgr キュー - ServerTestQWTLS2 の追加:

次のコード断片には、次のことが適用されます。

- ミニ認証サーバー・アドミニストレーターが、*認証要求 PIN*=12345678 を指定して ServerQMgr+ServerTesTQWTLS2 を追加し、ミニ認証サーバーを「サーバー (Server)」モードで開始したことを前提とします。
- MQePrivateClient クラスを使用して ServerQMgr をローカルに開始し、異なるバージョン MQePrivateServer2.ini (これは、 *PIN*、*KeyRingPassword*、および *CertReqPIN* のハードコーディング値を意図的に保持しない) を使用して、管理メッセージを作成し、さらにそれを使って ServerTestQWTLS2 キューを追加します。

```
{
    try{
       /* start ServerQMgr...                                              */
       String QMgrName       = "ServerQMgr";
       String QName          = "ServerTestQWTLS2"
       MQeAttribute qattr    = new MQeAttribute(
               new MQeWTLSCertAuthenticator(), new MQe3DESCryptor, null);
       String FileDesc       = "MsgLog:.";
       MQePrivateClient newC = new MQePrivateClient(
                                ".//MQePrivateServer2.ini",
                                "12345678",         /* or new PIN       */
                                "It_is_a_secret",   /* or new KeyRingPwd*/
                                 null);
       MQeQueueManager newQM =  newC.queueManager;
       /* create and use Admin msg to add ServerTestQWTLS2 queue        */
       MQeQueueAdminMsg msg  = new MQeQueueAdminMsg( );
       MQeFields parms       = new MQeFields( );
       msg.setTargetQMgr( QMgrName );
       msg.setName( QMgrName, QName );
       msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
       msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
       msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
       msg.putArrayOfByte( MQe.Msg_CorrelID,
                Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
       MQeFields msgTest = new MQeFields( );
       msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                              msg.getArrayOfByte( MQe.Msg_CorrelID ) );
       parms.putAscii( msg.Queue_QMgrName, QMgrName );
       parms.putAscii( msg.Queue_FileDesc, FileDesc );
       if ( qattr.getAuthenticator( ) != null ) /*add qattr auth details*/
         {
         parms.putAscii( msg.Queue_Authenticator,
                                    qattr.getAuthenticator( ).type( ) );
         if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
           {
           parms.putAscii( msg.Queue_AttrRule,
                                      "examples.rules.AttributeRule" );
           /* for the Queue to have its own credentials                  */
           parms.putByte( msg.Queue_TargetRegistry,
                                           msg.Queue_RegistryQueue );
           /* for the Queue to share its host QMgr's cresdentials        */
        // parms.putByte( msg.Queue_TargetRegistry,
        //                                   msg.Queue_RegistryQMgr );
           }
```

```
      }
    if ( qattr.getCryptor( ) != null )
      {
      parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
      if (! parms.contains( msg.Queue_AttrRule ) )
        parms.putAscii( msg.Queue_AttrRule,
                                  "examples.rules.AttributeRule" );
      }
    if ( qattr.getCompressor( ) != null )
      parms.putAscii( msg.Queue_Compressor,
                              qattr.getCompressor( ).type( ) );
    parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
    msg.create( parms );
    trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
    /* use Admin msg to add ServerTestQWTLS2                          */
    newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
    MQeAdminMsg respMsg      = null;
    trace(">>> Waiting for a response to create Admin Msg...");
    respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                              "AdminReplyQ", msgTest, null, 0, 3000);
    trace(">>> Admin Msg processed OK...");
    /* process Admin msg response                 ...                 */
    if ( respMsg == null )
      trace ( "i: create Queue failed, no response message received" );
    else
      {
      if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
        trace( "i: create Queue added queue OK..." );
      else
        trace( "i: create Queue failed: " + respMsg.getReason( ) );
      }
    newQM.close();
    }
  catch ( Exception e ) { e.printStackTrace(); }
  }
```

私用レジストリーを持つキュー・マネージャーを開始するキュー・ベースのセキュリティー

キュー・マネージャーおよびそのキューが認証可能なエンティティーである場合、つまり独自の証明書を持っている場合、これらの証明書にアクセスするには、キュー・マネージャーの開始時に適切なパラメーターが必要です。

適切な ini ファイルの Registry セクションでこれらのパラメーターをハード・コーディングすることは、ソリューションの開発中に便利なメカニズムである一方、実動システムには不適切です。可能な場合はいつでも、これらのパラメーターを対話的に収集し、ファイルに保管しないでキュー・マネージャー・インスタンスを開始するために使用してください。

MQePrivateClient クラスを使用し、パラメーターを渡す (MQePrivateClient2.ini ファイルのキーワードでハード・コーディングするのではなく) MQSeries Everyplace クライアン

ト・キュー・マネージャーの例が、例 234ページの『ClientQMgr キュー - XXX_service_reply キューの追加』で示されています。

キュー・ベースのセキュリティー - チャネルの再利用

データがキュー・マネージャーとリモート・キューの間に送信されると、キュー・マネージャーは、キューを所有しているリモート・キュー・マネージャーに対してチャネルをオープンします。デフォルトでは、リモート・キューが暗号機能などで保護されている場合、チャネルには、キューと全く同じレベルの保護が与えられます。並行してオープンするチャネルの数を少なくするために、キュー・マネージャーは、保護のレベルが適切であれば、既存のチャネルを再利用します。適切なレベルの保護を持ったチャネルが無い場合、キュー・マネージャーは、キューに必要な保護に合うように既存のチャネルの保護レベルを変更することもできます。キューとチャネルの両方において、属性ルールを使用して、デフォルト時の動作を変更することができます。これらのルールは、キュー (およびチャネル) における属性に適用され、キューのルールと同じではありません。

キューに対して属性ルールが定義されていると、キュー・マネージャーは、ルールを使用して、既存のチャネルがキューに十分な保護を持っているかどうか判別します。ルールの **equals()** メソッドにより `true` が戻されたら、そのチャネルを使用することができます。MQSeries Everyplace は、キューにおいて使用できるルール例、examples.rules.AttributeRule を提供しています。このルールでは、以下の条件が満たされている場合に、チャネルをキューに使用することができます。

- キューがオーセンティケーターを持っている場合、チャネルは同じオーセンティケーターを持っていなければならない。キューがオーセンティケーターを持っていない場合、チャネルはオーセンティケーターを持っていても持っていなくても構わない。
- キューが暗号機能を持っている場合、チャネルはキューに関して同じかまたはそれ以上の暗号機能を持っていなければならない。キューが暗号機能を持っていない場合、チャネルは暗号機能を持っていても持っていなくても構わない。
- キューまたはチャネルに圧縮機能が定義されているかどうかは関係ない。

ルール例が定義している「より強力な」暗号機能とは、次のとおりです。

- いかなる暗号機能も XOR と同じかまたはそれ以上である
- XOR 以外の暗号機能は、DES と同じかまたはそれ以上である
- それ以外の暗号機能 (Triple DES、RC4、RC6、および MARS) は、互いに同等であると見なされ、 XOR および DES 以上である

キューに適切な保護を持っている既存のチャネルが無い場合、キュー・マネージャーは、必要とされているレベルにアップグレードできるチャネルがあるかどうか調べます。チャネルに属性ルールが定義されている場合には、これを判別するために **permit()** メソッドが使用されます。examples.rules.AttributeRule は、以下の基準を使用します。

- チャネルが認証されている場合にはアップグレードできないが、認証を持っていない場合にはチャネルにオーセンティケーターを追加することができる。
- 暗号機能をチャネルに追加したり、強化する (前述の「より強力な」暗号機能の基準を使用) ことができる。暗号機能をチャネルから除去したり、現行以下のレベルの暗号機能に置き換えることはできない。
- 圧縮機能は変更でき、チャネルに追加したり、チャネルから除去することができる。

チャネルの再利用を許可する前に、宛先キューは、その現行の AttributeRule **equals()** メソッドを使って、チャネル属性が宛先キューに適切な保護レベルを提供できるかどうかを判別します。このことによって、ローカル・キュー・マネージャーと宛先キュー・マネージャーのキュー属性ルールに不整合が生じないようにします。

属性ルールは、管理メッセージを使用してキューが作成または変更される時にキューに設定されます。属性ルールは、キュー・マネージャー作成時に使用された構成ファイル内の *ChannelAttrRules* キーワードを使用して、チャネルに設定されます。

examples.rules.AttributeRule は実用的なデフォルトを提供していますが、ソリューションに特有のさまざまな動作が必要となる場合があります。デフォルトの examples.rules.AttributeRule を拡張するか、または希望する動作を定義したルールに置き換えて、チャネルの再利用方法を変更することができます。

ChannelAttrRules を設定せずに実行することはできますが、このモードでの操作は、お勧めしません。

メッセージ・レベルのセキュリティー

メッセージ・レベルのセキュリティーでは、発信側と受信側の MQSeries Everyplace アプリケーション間で、メッセージ・データの保護を容易にします。メッセージ・レベルのセキュリティーは、アプリケーション層サービスです。発信側の MQSeries Everyplace アプリケーションでメッセージ・レベルの属性を作成し、 **putMessage()** を使ってメッセージを宛先キューに入れるときに、その属性を提供する必要があります。受信側アプリケーションでは、getMessage を使用して、宛先キューからメッセージを取得するときにその属性を使えるように、適切で「一致する」メッセージ・レベルの属性をセットアップし、それを受信側のキュー・マネージャーに渡さなければなりません。

ローカル・セキュリティーの場合と同様に、メッセージ・レベルのセキュリティーでは、メッセージ (MQeFields オブジェクトの子孫) の属性のアプリケーションを活用します。発信側アプリケーションのキュー・マネージャーは、メッセージ・ダンプ・メソッドを使ってアプリケーションの **putMessage()** を処理します。そして、(付加される) 属性の **encodeData()** メソッドでメッセージ・データを保護します。受信側アプリケーションのキュー・マネージャーは、メッセージの 'restore' メソッドを使ってアプリケーションの **getMessage()** を処理します。そして、提供される属性の **decodeData()** メソッドで元のメッセージ・データを復元します。

使用法のシナリオ

メッセージ・レベルのセキュリティーは、通常、次の目的に役立ちます。

- 主に非同期キューを使用するために設計されるソリューション。
- アプリケーション・レベルのセキュリティーが重要なソリューション。つまり通常メッセージ・パスが、おそらく異なるプロトコルに接続される、複数のノードを介するフローを含むソリューション。メッセージ・レベルのセキュリティーは、慣例としてアプリケーション・レベルで承認を管理します。つまり、他の層のセキュリティーは不必要になります。

典型的なシナリオは、複数のオープン・ネットワークを介して送達される、ソリューション・サービスです。たとえば、最初から非同期操作が予期される、モバイル・ネットワークやインターネットを介する場合です。このシナリオでは、おそらくメッセージ・データは、異なるセキュリティー機能を持つ可能性のある複数のリンクを介して流れることも考えられますが、そのセキュリティー機能は、ソリューションの所有者によって制御されたり承認されたりするとは限りません。この場合、ソリューションの所有者は大抵、メッセージ・データの機密性の承認を中間メディアに委任しないで、承認管理を直接管理し、制御するほうがよいと考えます。

MQSeries Everyplace メッセージ・レベルのセキュリティーは、発信側および受信側アプリケーションに直接制御されている方法でのメッセージ・データの強度の保護を可能にし、終端から終端、アプリケーションからアプリケーションへのメッセージ・データ転送全体にわたる機密性を確実にする機能を、ソリューション設計者に提供します。

セキュア機能の選択

MQSeries Everyplace には、メッセージ・レベルのセキュリティーのために、次の 2 つの代替属性が備えられています。

MQeMAttribute

これは企業間通信に適しています。ここでは、相互の信用がアプリケーション層で厳しく管理されているので、承認された第三者による介入は必要ありません。利用できる MQSeries Everyplace のすべての対称暗号機能および圧縮機能はどれでも使うことができます。ローカル・セキュリティーと同じように、属性のキーが **putMessage()** および **getMessage()** でパラメーターとして提供される前に、それを事前設定しておく必要があります。メッセージ・レベルの保護のために、簡単かつ強力なメソッドが備えられています。これにより、公開鍵 PKI のオーバーヘッドを要することなく、強力な暗号機能を使ってメッセージの機密性を保護することができます。

MQeMTrustAttribute

これによって、ディジタル署名を使い、デフォルトの公開鍵 PKI を活用してディジタル・エンベロープ・スタイルの保護を提供する、さらに高機能なソリューションを提供します。これは、ISO9796 ディジタル署名 / 妥当性検査を使用し、受信側アプリケーションが、意図した送信側からメッセージが着信したことを証明できるようにします。提供された属性の暗号機能は、メッセージの

機密性を保護します。 SHA1 ダイジェストはメッセージの整合性を保ち、RSA 暗号機能は、意図した受信側だけがメッセージを復元できるようにします。 MQeMAttribute の場合と同様に、利用できる MQSeries Everyplace のすべての対称暗号機能および圧縮機能はどれでも使うことができます。サイズを最適化するために、使用される証明書は、WAP フォーラム WTLS 仕様で提案されている WTLS 仕様に基づいたミニ認証です。MQSeries Everyplace は、メッセージの暗号化および認証の必要性に応じて証明書を配布するために、デフォルトの公開鍵インフラストラクチャーを提供しています。

一般的な MQeMTrustAtribute 保護メッセージの形式は、次のとおりです。

```
RSA-enc{SymKey}, SymKey-enc {Data, DataDigest, DataSignature}
```

ここで、

RSA-enc:
意図した受信側の公開鍵によって、そのミニ認証から RSA 暗号化されている

SymKey:
疑似乱数対称キーが生成されている

SymKey-enc:
SymKey で対称的に暗号化されている

Data: メッセージ・データ

DataDigest:
メッセージ・データのダイジェスト

DigSignature:
メッセージ・データの発信側のディジタル署名

選択基準

MQeMAttribute は、キー・シードの内容を管理するソリューションの所有者に完全に依存しています。キー・シードとは、データの機密性を保護するために使う対称キーを派生させるために使用されるものです。このキー・シードは、発信側と受信側の両方のアプリケーションに提供する必要があります。キー・シードは、PKI を必要としないメッセージ・データの強度を保護するために単純なメカニズムを提供する一方で、効率的な操作管理はこのキー・シードに明らかに依存しています。

MQeMTrustAttribute は、MQSeries Everyplace のデフォルト PKI の利点を活用し、メッセージ・レベルの保護のディジタル・エンベロープ・スタイルを提供します。これは流れるメッセージ・データの機密性を保護するだけでなく、その整合性を検査し、発信側は意図した受信側だけがデータにアクセスできることを確実にします。また、受信側がデータの発信元の妥当性検査を行うことを可能にし、署名者が後でトランザクションの開始を否定できないようにもします。これを*拒否の取り消し* と言います。

メッセージ・データの終端間機密性を保護するだけのソリューションであれば、おそらく MQeMAttrribute で十分ですが、 1 対 1 (認証可能なエンティティーから認証可能なエンティティー) の転送およびメッセージ発信元の拒否の取り消しが重要なソリューションでは、MQeMTrustAttribute を選択すると良いでしょう。

使用法のガイド

次のコード・フラグメントは、MQeMAttribute および MQeMTrustAttribute を使って、メッセージを保護したり保護を解除したりする方法の例を示します。

MAttribute を使用した MQSeries Everyplace メッセージ・レベルのセキュリティー

```
   /*SIMPLE PROTECT FRAGMENT */
{
   MQeMsgObject msgObj      = null;
   MQeMAttribute attr = null;
   long confirmId           = MQe.uniqueValue();
   try{
       trace(">>>putMessage to target Q using MQeMAttribute"
              +" with 3DES Cryptor and key=my secret key");
        /* create the cryptor */
       MQe3DESCryptor tdes = new MQe3DESCryptor();
        /* create an attribute using the cryptor */
       attr = new MQeMAttribute(null,tdes,null );
        /* create a local key */
       MQeKey localkey = new MQeKey();
        /* give it the key seed */
       localkey.setLocalKey("my secret key");
        /* set the key in the attribute */
       attr.setKey(localkey );
        /* create the message */
       msgObj                    = new MQeMsgObject();
       msgObj.putAscii("MsgData","0123456789abcdef...");
        /* put the message using the attribute */
       newQM.putMessage(targetQMgrName, targetQName,
                  msgObj, attr, confirmId );
        trace(">>>MAttribute protected msg put OK...");
       }
    catch (Exception e)
       {
       trace(">>>on exception try resend exactly once...");
       msgObj.putBoolean(MQe.Msg_Resend, true );
       newQM.putMessage(targetQMgrName, targetQName,
                  msgObj, attr, confirmId );
       }
}

    /*SIMPLE UNPROTECT FRAGMENT */
{
       MQeMsgObject msgObj2     = null;
       MQeMAttribute attr2 = null;
       long confirmId2 = MQe.uniqueValue();
```

```
    try{
       trace(">>>getMessage from target Q using MQeMAttribute"+
          " with 3DES Cryptor and key=my secret key");
        /* create the attribute - we do not have to specify the cryptor, */
        /* the attribute can get this from the message itself  */
       attr2 = new MQeMAttribute(null,null,null );
        /* create a local key */
       MQeKey localkey = new MQeKey();
        /* give it the key seed */
       localkey.setLocalKey("my secret key");
        /* set the key in the attribute */
       attr2.setKey(localkey );
        /* get the message using the attribute */
       msgObj2 = newQM.getMessage(targetQMgrName, targetQName,
                                  null, attr2, confirmId2 );
       trace(">>>unprotected MsgData = "
          + msgObj2.getAscii("MsgData"));
       }
    catch (Exception e)
      {
     /*exception may have left */
    newQM.undo(targetQMgrName,            /*message locked on queue */
       targetQName, confirmId2 ); /*undo just in case */
    e.printStackTrace();               /*show exception reason */
      }
     ...
}
```

MTustAttribute を使用した MQSeries Everyplace メッセージ・レベルのセキュリティー

```
    /*SIMPLE PROTECT FRAGMENT */
{
    MQeMsgObject msgObj      = null;
    MQeMTrustAttribute attr = null;
    long confirmId           = MQe.uniqueValue();
try {
    trace(">>>putMessage from Bruce1 intended for Bruce8"
       + " to target Q using MQeMTrustAttribute with MARSCryptor ");
    /* create the cryptor */
    MQeMARSCryptor mars = new MQeMARSCryptor();
     /* create an attribute using the cryptor */
    attr = new MQeMTrustAttribute(null, mars, null);
     /* open the private registry belonging to the sender */
    String EntityName          = "Bruce1";
    String PIN                 = "12345678";
    Object Passwd              = "It_is_a_secret";
    MQePrivateRegistry sendreg = new MQePrivateRegistry();
    sendreg.activate(EntityName, ".//MQeNode_PrivateRegistry",
                                       PIN, Passwd, null, null );
     /* set the private registry in the attribute */
    attr.setPrivateRegistry(sendreg );
     /* set the target (recipient) name in the attribute */
    attr.setTarget("Bruce8");
     /* open a public registry to get the target's certificate */
```

```
|                    MQePublicRegistry pr = new MQePublicRegistry();
|                    pr.activate("MQeNode_PublicRegistry", ".//");
|                     /* set the public registry in the attribute */
|                    attr.setPublicRegistry(pr);
|                     /* set a home server, which is used to find the certificate*/
|                     /* if it is not already in the public registry */
|                    attr.setHomeServer(MyHomeServer +":8082");
|                     /* create the message */
|                    msgObj =new MQeMsgObject();
|                    msgObj.putAscii("MsgData","0123456789abcdef...");
|                     /* put the message using the attribute */
|                    newQM.putMessage(targetQMgrName, targetQName,
|                                     msgObj, attr, confirmId );
|                    trace(">>>MTrustAttribute protected msg put OK...");
|                    }
|                catch (Exception e)
|                    {
|                    trace(">>>on exception try resend exactly once...");
|                    msgObj.putBoolean(MQe.Msg_Resend, true);
|                    newQM.putMessage(targetQMgrName, targetQName,
|                                     msgObj, attr, confirmId );
|                    }
|                }
|
|                     /*SIMPLE UNPROTECT FRAGMENT */
|                {
|                    MQeMsgObject msgObj2     = null;
|                    MQeMTrustAttribute attr2 = null;
|                    long confirmId2 = MQe.uniqueValue();
|                try {
|                    trace(">>>getMessage from Bruce1 intended for Bruce8"
|                       + " from target Q using MQeMTrustAttribute with MARSCryptor ");
|                    /* create the cryptor */
|                    MQeMARSCryptor mars = new MQeMARSCryptor();
|                    /* create an attribute using the cryptor */
|                    attr2 = new MQeMTrustAttribute(null, mars, null);
|                    /* open the private registry belonging to the target */
|                       String EntityName          = "Bruce8";
|                       String PIN                 = "12345678";
|                       Object Passwd              = "It_is_a_secret";
|                    MQePrivateRegistry getreg = new MQePrivateRegistry();
|                    getreg.activate(EntityName, ".//MQeNode_PrivateRegistry",
|                                                      PIN, Passwd, null, null );
|                     /* set the private registry in the attribute */
|                    attr2.setPrivateRegistry(getreg);
|                     /* open a public registry to get the sender's certificate */
|                    MQePublicRegistry pr = new MQePublicRegistry();
|                    pr.activate("MQeNode_PublicRegistry", ".//");
|                     /* set the public registry in the attribute */
|                    attr2.setPublicRegistry(pr);
|                     /* set a home server, which is used to find the certificate*/
|                     /* if it is not already in the public registry */
|                    attr2.setHomeServer(MyHomeServer +":8082");
|                     /* get the message using the attribute */
|                    msgObj2 = newQM.getMessage(targetQMgrName,
```

```
                                    targetQName, null, attr2, confirmId2 );
    trace(">>>MTrustAttribute protected msg = "
          + msgObj2.getAscii("MsgData"));
    }
catch (Exception e)
    {
     /*exception may have left */
    newQM.undo(targetQMgrName,            /*message locked on queue */
            targetQName, confirmId2 ); /*undo just in case */
    e.printStackTrace();               /*show exception reason */
    }
}
```

拒否の取り消し

MQeMTrustAttribute は、メッセージにディジタル署名をします。これによって、受信側がメッセージの作成者の妥当性検査を行うことができ、作成者が後でメッセージの作成を否定できないようにします。これを *拒否の取り消し* と言います。この処理は、署名の妥当性検査を正常に行えるのは公開鍵 (証明書) だけであるという事実に基づいており、署名が該当する公開鍵を使用して作成されていることを検証します。作成者と断定された人がメッセージの作成を否定できる唯一の方法は、誰か他の人が秘密鍵に対するアクセス権を持っていたと主張するしかありません。

MQeMTrustAttribute を使用してメッセージが作成されると、送信者の私用レジストリーの秘密鍵を使用して、ディジタル署名が作成され、メッセージに送信者の名前が保管されます。メッセージが読み取られると (キュー・マネージャーの **getMessage()** メソッドを使用する)、送信者の公開証明書を使用して、ディジタル署名の妥当性検査が行われます。メッセージは、メッセージが送信者としてのメッセージに保管されている名前のエンティティーによって作成されたものであることが判明して、署名の妥当性検査が正常に完了した場合にだけ、正常に読み取られます。

MQeMTrustAttribute がキュー・マネージャーの **getMessage()** メソッドに対するパラメーターとして指定されると、属性はディジタル署名の妥当性検査を行いますが、メッセージがユーザーのアプリケーションに戻されるまでに、署名に関するすべての情報が破棄されます。拒否の取り消しが重要な場合には、この情報の記録を保持しておく必要があります。これを最も簡単に行うには、暗号化されたメッセージのコピーを保持しておきます。コピーには、ディジタル署名が含まれているからです。これは、属性を使用せずに **getMessage()** メソッドを使って行えます。暗号化されたメッセージが戻されるので、これをローカル・キューなどに保管します。メッセージの内容にアクセスするための属性を適用して、メッセージの暗号化を解除することができます。

次に、これを行う方法の例としてのコード・フラグメントを示します。

暗号化されたメッセージのコピーを保管する

```
/*SIMPLE FRAGMENT TO SAVE ENCRYPTED MESSAGE*/
{
MQeMsgObject msgObj2 = null;
MQeMTrustAttribute attr2 = null;
```

```
long confirmId2 = MQe.uniqueValue();
long confirmId3 = MQe.uniqueValue();
try {
    trace(">>>getMessage from Bruce1 intended for Bruce8"
  + " from target Q using MQeMTrustAttribute with MARSCryptor ");
    /* read the encrypted message without an attribute */
    MQeMsgObject tmpMsg1 = newQM.getMessage(targetQMgrName,
    targetQName, null, null, confirmId2 );
    /* save the encrypted message - we cannot put it directly */
    /* to another queue because of the origin queue manager  */
    /* data. Embed it in another message                      */
    MQeMsgObject tmpMsg2 = new MQeMsgObject();
    tmpMsg2.putFields("encryptedMsg", tmpMsg1);
    newQM.putMessage(localQMgrName, archiveQName, tmpMsg2, null, confirmId3);
    trace(">>>encrypted message saved locally");
    /* now decrypt and read the message ... */
    /* create the cryptor */
    MQeMARSCryptor mars = new MQeMARSCryptor();
    /* create an attribute using the cryptor */
    attr2 = new MQeMTrustAttribute(null, mars, null);
    /* open the private registry belonging to the target */
    String EntityName          = "Bruce8";
    String PIN                 = "12345678";
    Object Passwd              = "It_is_a_secret";
    MQePrivateRegistry getreg = new MQePrivateRegistry();
    getreg.activate(EntityName, ".//MQeNode_PrivateRegistry",
PIN, Passwd, null, null );
    /* set the private registry in the attribute */
    attr2.setPrivateRegistry(getreg);
    /* open a public registry to get the sender's certificate */
    MQePublicRegistry pr = new MQePublicRegistry();
    pr.activate("MQeNode_PublicRegistry", ".//");
    /* set the public registry in the attribute */
    attr2.setPublicRegistry(pr);
    /* set a home server, which is used to find the certificate*/
    /* if it is not already in the public registry */
    attr2.setHomeServer(MyHomeServer +":8082");
    /* decrypt the message by unwrapping it */
    msgObj2 = tmpMsg1.unwrapMsgObject(attr2);
    trace(">>>MTrustAttribute protected msg = "
+ msgObj2.getAscii("MsgData"));

catch (Exception e)
  { /*exception may have left */
    newQM.undo(targetQMgrName, /*message locked on queue */
       targetQName, confirmId2 ); /*undo just in case */
    e.printStackTrace(); /*show exception reason */
   }
}

P187
private registry service
2nd paragraph, 2nd line, remove "an" from the end:
" ... dependent services as"
```

私用レジストリー・サービス

このセクションでは、MQSeries Everyplace が提供する私用レジストリー・サービスを説明します。

私用レジストリーと認証可能エンティティーの概念

相互認証に基づくミニ認証を使うキュー・ベースのセキュリティーと、ディジタル署名を使うメッセージ・レベルのセキュリティーでは、認証可能なエンティティーという概念が提示されています。相互認証というと、一般に 2 人のユーザー間での認証を思い浮かべますが、実際にはメッセージングにはユーザーという概念はありません。メッセージング・サービスの通常のユーザーはアプリケーションで、これらがユーザー概念を扱います。

MQSeries Everyplace は認証の宛先の概念をユーザー (人) から認証可能なエンティティーに抽象化します。このことは、認証可能なエンティティーが人間である可能性を除外するものではありませんが、アプリケーションによって選択されて割り当てられることになります。

内部的には、MQSeries Everyplace はすべてのキュー・マネージャーを定義します。これらは、認証可能なエンティティーとして、発信側の可能性もありますし、ミニ認証に従属するサービスの宛先の可能性もあります。また、MQSeries Everyplace では、認証プログラムに基づいてミニ認証を使うよう定義されたキューを、認証可能なエンティティーとしても定義します。したがって、これらのサービスをサポートするキュー・マネージャーは、1 つの認証可能なエンティティー (キュー・マネージャーだけ) か、一群の認証可能なエンティティー (キュー・マネージャーと、証明書に基づく認証プログラムを使うそれぞれのキュー) を持つことができます。

MQSeries Everyplace は、構成可能なオプションを提供して、キュー・マネージャーとキューが、認証可能なエンティティーとして自動登録することを可能にします。 MQSeries Everyplace 私用レジストリー・サービス (MQePrivateRegistry) は、 MQSeries Everyplace アプリケーションが認証可能なエンティティーを自動登録し、結果の証明書を管理するためのサービスを提供します。

認証可能なエンティティーとして登録されたアプリケーションはすべて、MQeMTrustAttribute を使用して保護されたメッセージ・レベルのサービスの発信側としても受信側としても使うことができます。

私用レジストリーと認証可能エンティティーの証明書

便利なことに、認証可能なエンティティーごとにそれぞれの信任状が必要とされています。これにより 2 つの問題点が生じます。 1 つ目は、証明書を入手するためにどのように登録を実行するかということ、そして 2 つ目は、その証明書を安全な方法でどこに管理するかということです。 MQSeries Everyplace 私用レジストリー・サービスは、こ

れらの 2 つの問題を解決するのに役立ちます。これらのサービスを使えば、安全な方法でその証明書を作成する認証可能なエンティティーの自動登録を起動すると同時に、安全なリポジトリーを提供できます。

私用レジストリー (基本レジストリーの子孫) は、基本レジストリーに保護または暗号トークンの特徴の機能を追加します。たとえば、公開オブジェクト (ミニ認証) と私用オブジェクト (秘密鍵) 用の保護リポジトリーになります。また、許可ユーザーに、私用オブジェクトへのアクセスを制限するメカニズムを提供します。さらに、私用オブジェクトが私用レジストリーから漏れないようにするサービス (たとえば、ディジタル署名、RSA 暗号化解除など) をサポートしています。また、共通インターフェースを提供することによって、基礎となるデバイス・サポートを見分けることができないようにします。

自動登録

MQSeries Everyplace には、自動登録をサポートするデフォルトのサービスがあります。これらのサービスは、認証可能なエンティティーが構成されると、自動的に起動します。たとえば、キュー・マネージャーの開始時、新しいキューの定義時、さらに MQSeries Everyplace アプリケーションが MQePrvateRegistry を直接使用して、認証可能なエンティティーを新しく作成する場合などです。登録が起動すると、新しい証明書が作成されて、認証可能なエンティティーの私用レジストリーに格納されます。自動登録のステップには、新しい RSA キーのペアを生成すること、秘密鍵を私用レジストリーで保護し、そこに保管すること、そして、デフォルトのミニ認証サーバーに対する新しい認証要求に公開鍵をパッケージすることが含まれます。ミニ認証サーバーが構成されて使用可能になっており、認証可能なエンティティーが (証明書を持つ権限を持つ) ミニ認証サーバーによって事前登録されている場合、ミニ認証サーバーはすでに所有しているミニ認証と共に、認証可能なエンティティーの新しいミニ認証を戻し、これらの認証は、エンティティーの新しい証明書として、保護された秘密鍵と一緒に認証可能なエンティティーの私用レジストリーに格納されます。

自動登録は、認証可能なエンティティーの証明書を設定する簡単なメカニズムですが、メッセージ・レベルの保護をサポートするために、エンティティーは、独自の証明書 (ディジタル署名を容易にする) と、意図した受信側の公開鍵 (ミニ認証) にアクセスする必要もあります。

使用法のシナリオ

MQSeries Everyplace の私用レジストリーの基本的な目的は、MQSeries Everyplace の認証可能なエンティティーの証明書に私用リポジトリーを提供することです。認証可能なエンティティーの証明書は、エンティティーのミニ認証 (エンティティーの公開鍵をカプセル化する) と、およびエンティティーの (鍵リングで保護された) 秘密鍵から成っています。

典型的な使用法のシナリオは、他の MQSeries Everyplace セキュリティー機能に関連付けて考慮する必要があります。

MQeWTLSCertAuthenticator を使用するキュー・ベースのセキュリティー

キュー・ベースのセキュリティーが使用されており、キュー属性が MQeWTLSCertAuthenticator で定義されている場合 (ミニ認証ベースの相互認証) は、関係する認証可能なエンティティーは MQSeries Everyplace に所有されています。そのようなキューでメッセージにアクセスするのに使用されるキュー・マネージャー、そのようなキューを所有するキュー・マネージャー、およびキュー自体は、すべて認証可能なエンティティーであり、独自の証明書を持つ必要があります。正しい構成オプションを使用し、MQSeries Everyplace ミニ認証発行サービスのインスタンスを使用することによって、キュー・マネージャーとキューの作成時に自動登録が起動され、新しい証明書を作成してエンティティーの独自のレジストリーに保管します。

MQeMTrustAttribute を使用するメッセージ・レベルのセキュリティー

メッセージ・レベルのセキュリティーを MQeMTrustAttribute で使用する場合はいつでも、MQeMTrustAttribute で保護されたメッセージの発信側と受信側は、独自の証明書を持っている必要のある、アプリケーションが所有する認証可能なエンティティーです。この場合、アプリケーションは MQePrivateRegistry のサービス (および MQSeries Everyplace ミニ認証発行サービスのインスタンス) を使って、自動登録を起動し、エンティティーの証明書を作成してエンティティーの独自の私用レジストリーに保管する必要があります。

セキュア機能の選択

MQSeries Everyplace バージョン 1 では、認証可能なエンティティーの証明書の代替セキュア・リポジトリーをサポートしません。 MQeWTLSCertAuthenticator を使用するキュー・ベースのセキュリティー、または MQeMTrustAttribute を使用するメッセージ・レベルのセキュリティーが使用される場合、私用レジストリー・サービスを使う必要があります。

選択基準

私用レジストリーの選択基準は、キュー・ベースおよびメッセージ・レベルのセキュリティーの選択基準と同じです。

使用法のガイド

キュー・ベースのセキュリティーを使用する前に、MQSeries Everyplace が所有する認証可能なエンティティーには証明書がなければなりません。これは、正しい構成を完了して、キュー・マネージャーの自動登録を起動することによって行われます。それには次のステップが必要です。

1. MQSeries Everyplace ミニ認証発行サービスのインスタンスをセットアップし、開始します。
2. 「管理 (Administration)」モードで、キュー・マネージャーの名前を有効な認証可能エンティティーとして追加し、さらにエンティティーの一時使用の認証要求 PIN を追加します。

3. 「サーバー (Server)」モードでミニ認証サーバーを開始します。
4. MQePrivateClient1.ini および MQePrivateServer1.ini を、「SimpleCreateQM を使用して ClientQMgr および ServerQMgr インスタンスを作成する」の説明に従って構成し、キュー・マネージャーが SimpleCreateQM を使って作成されるときに自動登録が起動するようにします。このセクションでは、ini ファイルの Registry セクションで必要なキーワード、およびエンティティーの一時使用の認証要求 PIN を使用する位置を説明します。

メッセージ・レベルのセキュリティーを使用して、MQeMTrustAttribute を使ってメッセージを保護する前に、アプリケーションは私用レジストリー・サービスを使って、発信側および受信側エンティティーが必ず証明書を持つようにする必要があります。それには次のステップが必要です。

1. MQSeries Everyplace ミニ認証発行サービスのインスタンスをセットアップし、開始します。
2. 「管理 (Administration)」モードで、アプリケーション・エンティティーの名前を追加し、エンティティーを一時使用の認証要求 PIN に割り振ります。
3. 「サーバー (Server)」モードでミニ認証サーバーを開始します。
4. 以下のプログラム断片と同様のプログラムを使って、アプリケーション・エンティティーの自動登録を起動します。これによって、エンティティーの証明書が作成され、私用レジストリーに保管されます。

```
/* SIMPLE MQePrivateRegistry FRAGMENT                                */
try
   {
   /* setup PrivateRegistry parameters                               */
   String EntityName       = "Bruce";
   String EntityPIN                = "11111111";
   Object KeyRingPassword  = "It_is_a_secret";
   Object CertReqPIN       = "12345678";
   Object CAIPAddrPort             = "9.20.X.YYY:8082";
   /* instantiate and activate a Private Registry.                   */
   MQePrivateRegistry preg = new MQePrivateRegistry( );
   preg.activate( EntityName,        /* entity name                 */
                  ".//MQeNode_PrivateRegistry", /* directory root  */
                  EntityPIN,         /* private reg access PIN      */
                  KeyRingPassword,   /* private credential keyseed */
                  CertReqPIN,        /* on-time-use Cert Req PIN    */
                  CAIPAddrPort );    /* addr and port MiniCertSvr   */
   trace(">>> PrivateRegistry activated OK ...");
   }
catch (Exception e)
   {
   e.printStackTrace( );
   }
```

公開レジストリー・サービス

このセクションでは、MQSeries Everyplace が提供する公開レジストリー・サービスを説明します。

MQSeries Everyplace には、MQSeries Everyplace ノード間で認証可能なエンティティーの*公開信任状* (ミニ認証) の共用を容易にする、デフォルトのサービスがあります。これらのミニ認証へのアクセスは、メッセージ・レベルのセキュリティーの前提条件です。MQSeries Everyplace 公開レジストリー (および基本レジストリーの子孫) は、ミニ認証へ公的にアクセスできるリポジトリーを提供します。これは、携帯電話の個人電話番号登録簿サービスに似ています。異なる点は、電話番号の代わりに、認証可能なエンティティーのミニ認証セットが使われる点です。MQSeries Everyplace 公開レジストリーは純粋な受動サービスではありません。保持していないミニ認証を提供するようにアクセスするときに、公開レジストリーが有効なホーム・サーバーで構成されている場合には、その公開レジストリーは、ホーム・サーバーの公開レジストリーから、要求されたミニ認証を自動的に入手しようとします。また、ミニ認証を他の MQSeries Everyplace ノードの公開レジストリーと共用するためのメカニズムも備わっています。これらのサービスがまとまって、自動化されたインテリジェントなミニ認証複製サービスを構成します。これにより、必要なときに必要なミニ認証を使うことができます。

使用法のシナリオ

公開レジストリーの典型的なシナリオは、特定の MQSeries Everyplace ノードの公開レジストリーが、最も頻繁に必要とされるミニ認証が使用されるときにそれを保管して構築するように、これらのサービスを使用することです。

この単純な例は、MQSeries Everyplace ホーム・サーバーから必要な他の認証可能なエンティティーのミニ認証を自動的に入手し、その後、それらを公開レジストリーに保管するように MQSeries Everyplace クライアントをセットアップすることです。

セキュア機能の選択

異なる MQSeries Everyplace ノードの公開レジストリー間でミニ認証を共用し、それを入手するための公開レジストリーのアクティブな機能を使用するかどうかは、ソリューション作成者の選択に任されています。

このインテリジェントな複製の代替方法は、帯域外ユーティリティーに、必要なすべてのミニ認証を含む MQSeries Everyplace ノードの公開レジストリーを初期化させてから、それらを使用するすべてのセキュア・サービスを使用可能にすることです。

選択基準

MQSeries Everyplace ノードの公開レジストリーで使用可能なミニ認証セットの帯域外初期化には、ソリューションが主に非同期で MQSeries Everyplace ノードのホーム・サーバーへの同期接続が困難な場合に、公開レジストリーのアクティブな機能を使用するよりも利点の多いことがあります。しかし、この接続が使用可能であると考えられる場

合、公開レジストリーのアクティブなミニ認証複製サービスは、MQSeries Everyplace ノード公開レジストリーで最も役立つミニ認証セットを自動的に保持するのに役立つツールです。

使用法のガイド

```
/*SIMPLE MQePublicRegistry shareCertificate FRAGMENT */
try {
String EntityName = "Bruce";
String EntityPIN = "12345678";
Object KeyRingPassword = "It_is_a_secret";
Object CertReqPIN = "12345678";
Object CAIPAddrPort = "9.20.X.YYY:8082";
/*instantiate and activate PublicReg */
MQePublicRegistry pubreg = new MQePublicRegistry();
pubreg.activate("MQeNode_PublicRegistry",".\\");
/* auto-register Bruce1,Bruce2...Bruce8 */
/* ... note that the mini-certificate issuance service must */
/* have been configured to allow the auto-registration    */
for (int i = 1; i < 9; i++)
{
EntityName = "Bruce"+(new Integer(i)).toString();
MQePrivateRegistry preg = new MQePrivateRegistry();
/* activate() will initiate auto-registration */
preg.activate(EntityName, ".\\MQeNode_PrivateRegistry",
EntityPIN, KeyRingPassword, CertReqPIN, CAIPAddrPort);
/* save MiniCert from PrivReg in PubReg*/
pubreg.putCertificate(EntityName,
preg.getCertificate(EntityName ));
/*before share of MiniCert */
pubreg.shareCertificate(EntityName,
preg.getCertificate(EntityName ),"9.20.X.YYY:8082");
preg.close();
}
pubreg.close();
}
catch (Exception e)
{
e.printStackTrace();
}
```

注:

1. レジストリー・インスタンスは複数回活動化することができないため、上記の例では、私用レジストリーのアクセスの推奨例を示しています。つまり、MQePrivateRegistry の新しいインスタンスを作成し、そのインスタンスを活動化してから、必要な操作を実行してインスタンスをクローズするという方法です。
2. ホーム・サーバーで公開レジストリーを使用して証明書を共用したい場合は、公開レジストリーは MQeNode_PublicRegistry と呼ばれるものでなければなりません。

ミニ認証発行サービス

MQSeries Everyplace には、デフォルトのミニ*認証発行*サービス が組み込まれています。これは、私用レジストリー自動登録要求を満たすように構成できます。付属するツールを使うことにより、特定のソリューションで、ミニ認証発行サービスを設定して管理し、こうして注意深く制御された一群のエンティティー名にミニ認証を発行することができます。この発行サービスには、以下のような特性があります。

- 登録された認証可能なエンティティーのセットの管理
- ミニ認証の発行 (ミニ認証 は、WAP WTLS 仕様に準拠しています)
- ミニ認証リポジトリーの管理

付属するツールを使うことにより、ミニ認証発行サービスの管理者は、エンティティー名と登録アドレスを登録し、一時使用の*認証要求 PIN* を定義することにより、エンティティーへのミニ認証発行を許可することができます。通常、これは要求側の認証性の妥当性検査をオフラインで実行した後に行われれます。認証要求 PIN は、意図したユーザーに通知できます (新しいキャッシュ・カードが発行されるときに、キャッシュ・カード PIN が通知されるのと同様です)。次に私用レジストリーのユーザー (たとえば、MQSeries Everyplace アプリケーションや MQSeries Everyplace キュー・マネージャー) を構成し、起動時にこの認証要求 PIN を提供するように設定することができます。私用レジストリーが自動登録を起動すると、ミニ認証発行サービスでは、生成される新しい認証要求を妥当性検査し、新しいミニ認証を発行してから、登録済みの認証要求 PIN をリセットして、それを再利用できないようにします。新しいミニ認証要求の自動登録はすべて、安全なチャネル経由で処理されます。

ミニ認証発行サービスによって発行されているミニ認証は、発行サービスのレジストリーに保管されます。ミニ認証が再発行されると (たとえば、有効期限切れなどで)、有効期限が切れたミニ認証は保存されます。

ミニ認証発行サービス・サーバーのインスタンスの構成、開始、および終了

MQSeries EveryplaceMiniCertificateServer.ini を使用する構成

MQeMiniCertificateServer.ini は構成ファイルの例です。この例を変更し、MQeMiniCertificateServer 起動時にそれを使用することによって、MQeMiniCertificateServer のインスタンスを作成できます。MQeMiniCertificateServer.ini には、Alias、ChannelManager、Listener および MiniCertSvrRegistry セクションが含まれます。 MQeMiniCertificateServer のインスタンスは、動作の自動構成の起動時に、これらのセクションの内容を使用します。

MQeMiniCertificateServer.ini は、ExamplesMQeServer.ini を拡張したものです。拡張については、ここで説明されています。その他のオプションについては、ExamplesMQeServer.ini の説明を参照してください。

[Alias] セクションの拡張

次の 2 つの必須キーワードが追加されます。

MiniCertSvrRegistry

この設定は、使用されるレジストリーのクラス名を識別します。

MiniCertIssuanceManager

この設定は、MQeMiniCertIssuanceInterface をインプリメントするクラスの名前を識別します。

追加の [MiniCertServerRegistry] セクション

このセクションには、次に 2 つのオプション・キーワードが含まれます。

InitialPIN

これは、有効な MQeMiniCertificateServer Administrator の PIN を識別します。これは、MQeMiniCertificateServer によって、その私用レジストリーへのアクセスを活動化し、入手するために使用されるものです。

KeyRingPassword

これは、MiniCertificateServer の私用レジストリーに保管される私用オブジェクトを保護するために使用される、パスワードまたはパスフレーズを識別します。

MQeMiniCertificateServerGUI の開始

MQeMiniCertificateServerGUI.bat は、起動ファイルの例です。MQeMiniCertificateServer のインスタンスは、この例を変更して使用することによって開始できます。この例では次のコマンドを使用します。

```
java  com.ibm.mqe.server.MQeMiniCertificateServer <parameter1> <parameter2>
```

ここで、

<parameter1> = com.ibm.MQe.Server.MCSMessageBundle
(または MQeMiniCertificateServer メッセージの
ListResourceBundle を変換したバージョン)
<parameter2> = Examples.Trace.MQeTraceResource
(または MQSeries Everyplace ベース・メッセージの
ListResourceBundle を変換したバージョン)

GUI を使用してミニ認証発行サービスを初めて開始する

MQeMiniCertificateServerGUI.bat を呼び出すと、次のような結果が表示されます。

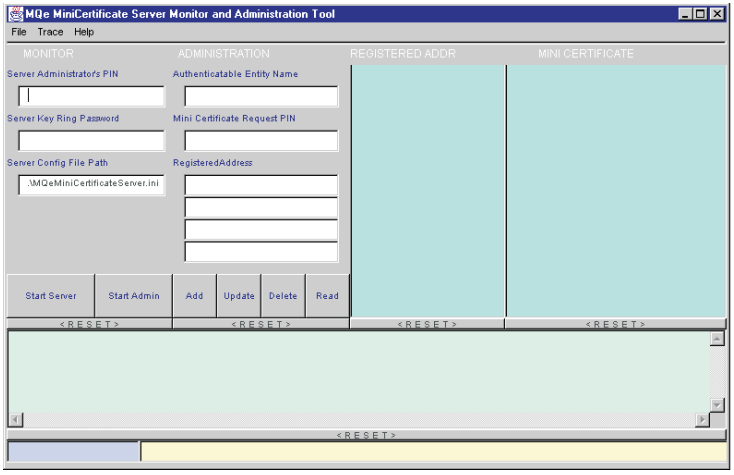

図 37. ミニ認証サーバー GUI

ミニ認証サーバーサーバーを初めて起動する場合には、管理者は次のことを行う必要があります。

1. 「サーバー管理者の PIN (*ServerAdministrator's PIN*)」フィールドに、ミニ認証サーバーのこのインスタンスへのアクセスに使用する PIN を入力します (ここでは、'12345678' になっています)。
2. 管理者がミニ認証サーバーのレジストリーの私用オブジェクトを保護するために使用するパスワードまたはパスフレーズを、「サーバー・キー・リング・パスワード (*ServerKey Ring Password*)」フィールドに入力します (ここでは、'It_is_a_secret' になっています)。
3. 起動構成ファイルのパスおよびファイル名を、「サーバー構成ファイル・パス (*Server Config File Path*)」フィールドに入力します (ここでは './MQeMiniCertificateServer.ini' になっています)。
4. 「サーバーの開始 (*Start Server*)」ボタンをクリックします。

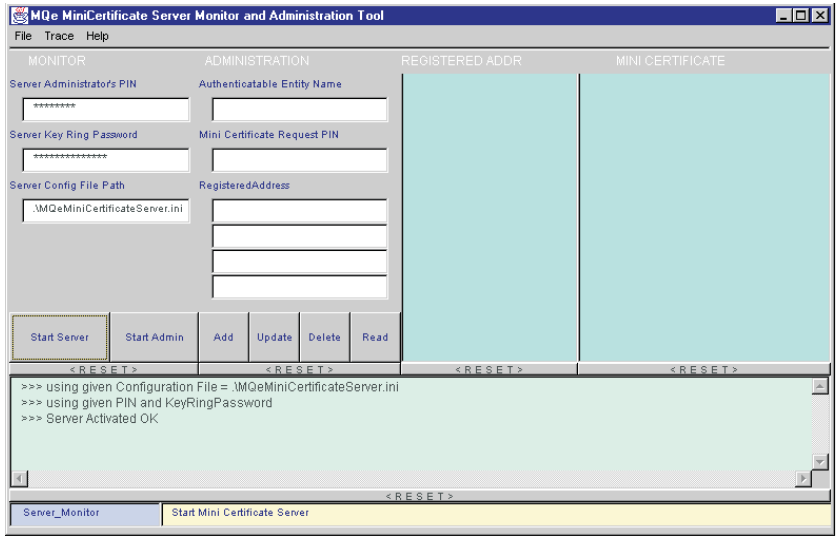

図 38. 開始されたミニ認証サーバー

注: GUI の下部の左側にある「モード (*Mode*)」標識は、サーバーが開始したことを 'Server_Monitor' と表示して示します。モード標識の右側の「コンテキスト (Context)」出力は、「サーバーの開始 (Start Server)」ボタンのコンテキスト・ヘルプを示します。「モード (*Mode*)」および「コンテキスト (*Context*)」の上にある「モニター (Monitor)」出力は、有効なモニター出力の例です。

管理ツールの使用

管理モードの開始

管理ツールを使用するために、MQeMiniCertificateServerGUI を起動し、「管理 (Administration)」モードを開始する必要があります。
これは、MQeMiniCertificateServerGUI.bat を起動し、「サーバー管理者の PIN (*Server Administrator's PIN*)」、「サーバー・キー・リング・パスワード (*ServerKey Ring Password*)」、および「サーバー構成ファイル・パス (*Server Config File Path*)」入力フィールドに入力してから、「管理の開始 (*Start Admin*)」ボタンを選択することによって行えます。このタスクからの視覚的フィードバックの例を以下に示します。

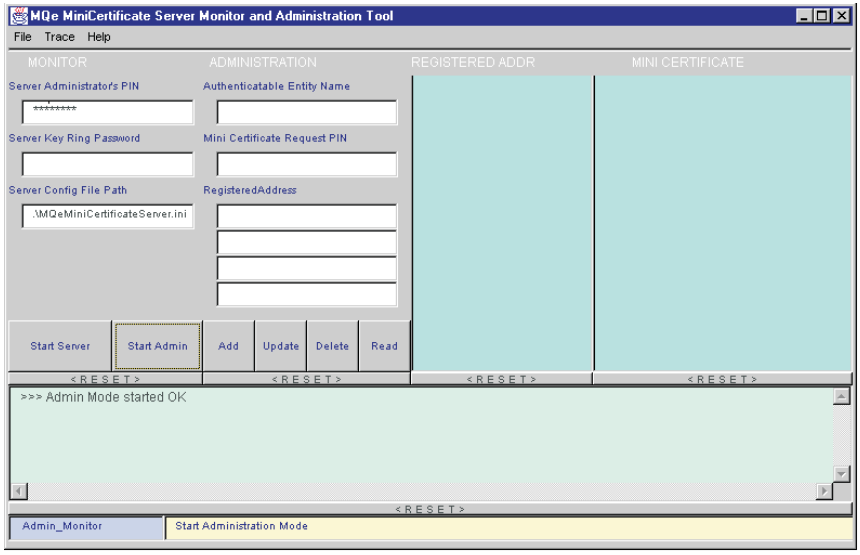

図 39. ミニ認証サーバーの管理モード

新しい認証可能なエンティティーを追加する

「管理 (Administration)」モードを開始してあれば、新しい認証可能なエンティティーを追加するには、適切な入力フィールドにエンティティーの名前とアドレスを指定し、一時使用の認証要求 PIN を設定してから、「追加 (*Add*)」ボタンをクリックします。このタスクからの視覚的フィードバックの例を以下に示します。

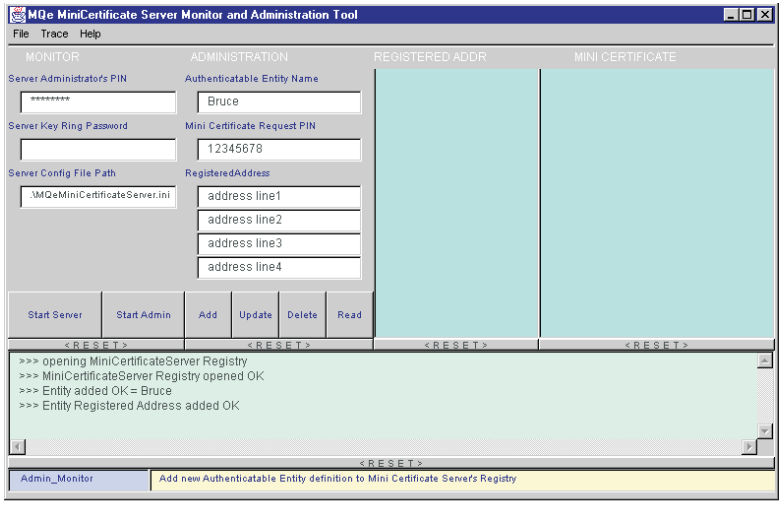

図 40. 新しい認証可能なエンティティーを追加する

認証可能なエンティティーを更新する

登録済みの認証可能なエンティティーの詳細を更新することは、エンティティーの追加と類似しています。「管理 (Administration)」モードになっていれば、認証可能なエンティティーを更新した詳細が提供されます。これには、新しい認証要求 PIN が含まれていることもあります (該当する場合)。その後、「更新 (*Update*)」ボタンをクリックして更新します。このタスクからの視覚的フィードバックの例を以下に示します。

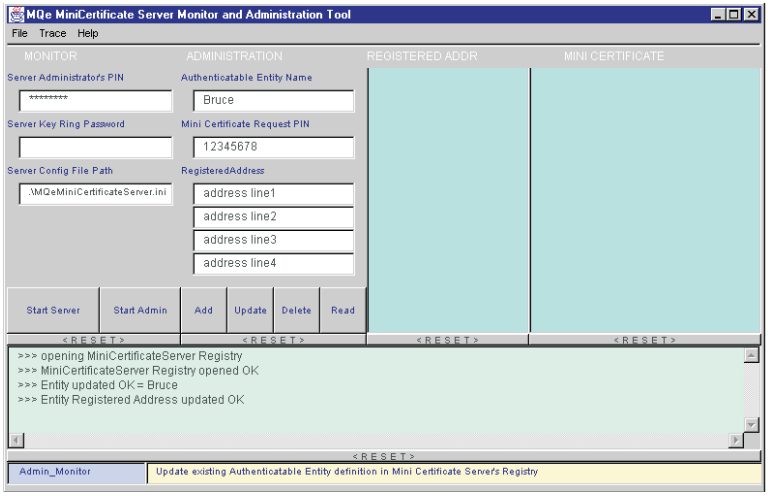

図 41. 認証可能なエンティティーを更新する

認証可能なエンティティーを削除する

登録済みの認証可能なエンティティーの詳細の削除は、認証可能なエンティティーの名前を入力フィールドに入力してから、「削除 (*Delete*)」ボタンをクリックすることによって実行できます。

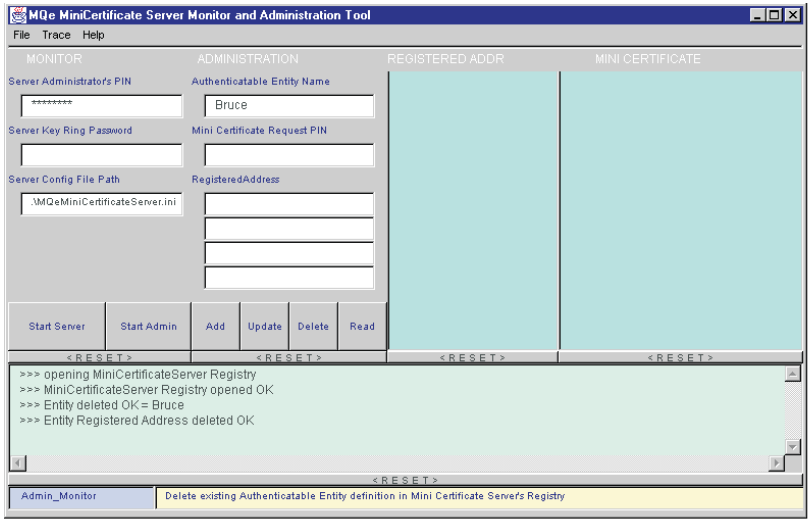

図 42. 認証可能なエンティティーを削除する

認証可能なエンティティーの詳細を読み取る

登録済みの認証可能なエンティティーの詳細を読み取るには、認証可能なエンティティーの名前を入力フィールドに入力してから、「読み取り (*Read*)」ボタンをクリックします。このタスクからの視覚的フィードバックの例を以下に示します。

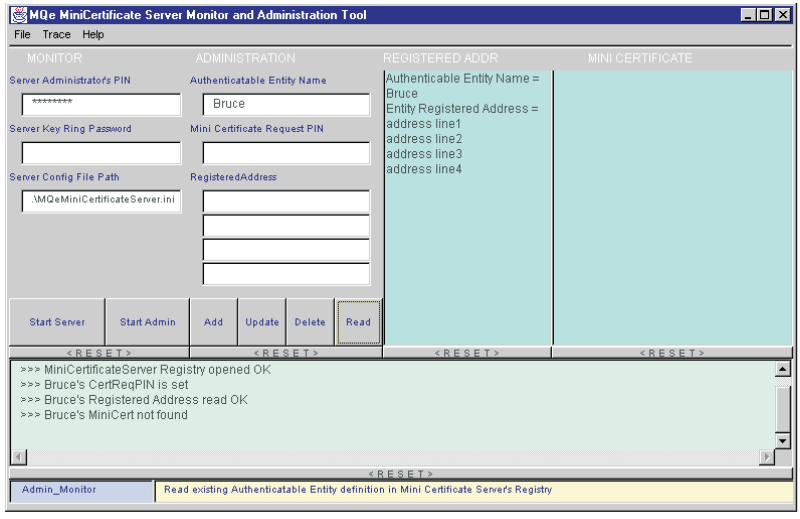

図 43. 認証可能なエンティティーを読み取る

これは、登録済みの認証可能なエンティティーの詳細を表示するメソッドを提供します。視覚的フィードバックは、登録済みアドレスとミニ認証、および一時使用の認証要求 PIN の状況 (使用可能な場合) を表示します。通常の使用では、認証可能なエンティティーは登録された後からミニ認証が発行される前までは、登録済みアドレスが表示され、認証要求 PIN の状況が設定されますが、ミニ認証状況は見つかりません。ミニ認証が発行された後は、登録済みアドレスと現行のミニ認証が表示され、要求 PIN 状況は設定されません。

「ファイル (File)」メニューの「オープン (Open)」オプションの使用

名前の入力を必要としない認証可能なエンティティーを選択するために、「読み取り (*Read*)」に加えて「オープン (*Open*)」オプションが提供されています。このオプションを使用するには、「管理 (Administration)」モードで次のように実行します。

1. 「ファイル (*File*)」プルダウン・メニューから、「オープン (*Open*)」オプションを選択する。

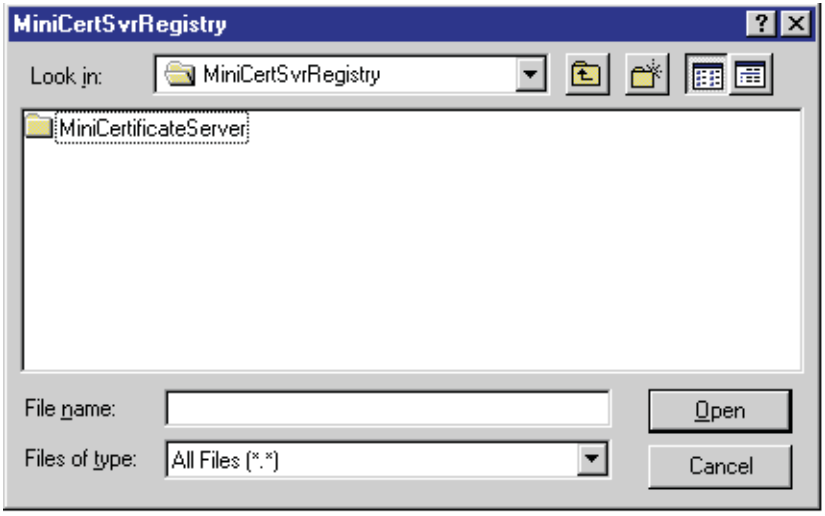

図 44. MQSeries Everyplace の認証可能なエンティティーの詳細の表示

2. 表示されたリストから、「*EntityAddr*」フォルダーを選択し、「オープン (*Open*)」ボタンをクリックする。

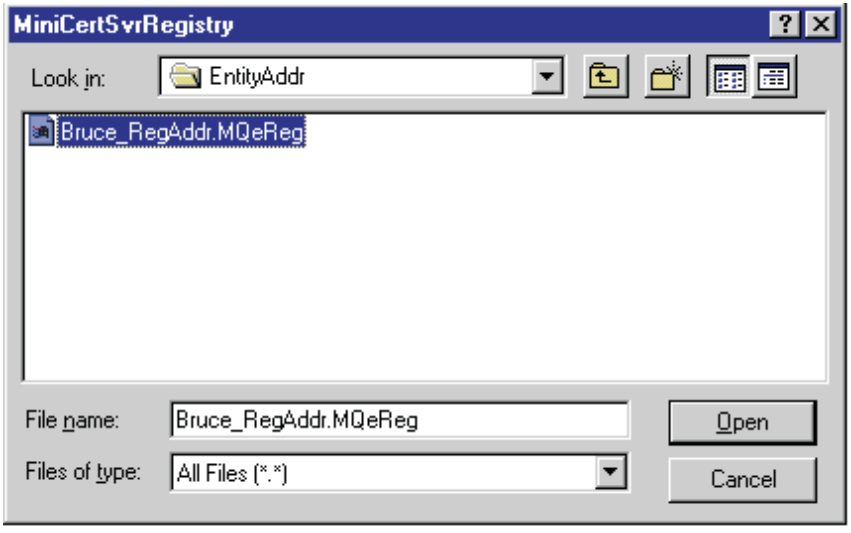

図 45. MQSeries Everyplace の認証可能なエンティティーの詳細の表示

3. 表示されたリストから、照会したいエンティティーの名前を選択して、「オープン (*Open*)」ボタンをクリックする。

エンティティーの詳細は、図46 で示されるように表示されます。

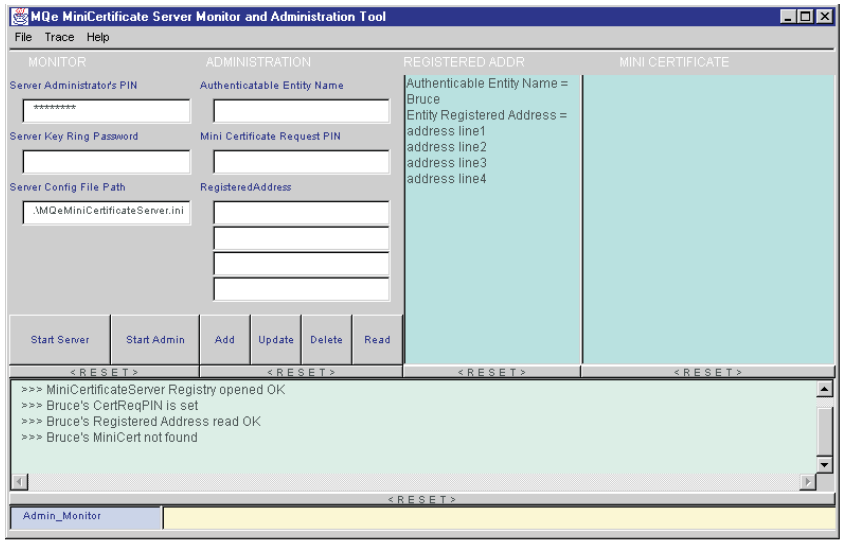

図 46. MQSeries Everyplace の認証可能なエンティティーの詳細の表示

操作

開始と停止

MQeMiniCertificateServerGUI のインスタンスの開始、および GUI を使用したサーバーの開始または「管理 (Administration)」モードの開始については、 263ページの『MQeMiniCertificateServerGUI の開始』および 265ページの『管理モードの開始』 で説明されています。どちらの場合も、 MQeMiniCertificateServerGUI インスタンスを終了するには、「ファイル (*File*)」プルダウン・メニューで「終了 (*Exit*)」オプションを選択します。確認ダイアログで「はい」を選択し、ミニ認証サーバーのシャットダウンを完了します。

モニターとログ記録

「サーバー・モニター (Server_Monitor)」モードまたは「管理モニター (Admin_Monitor)」モードでサーバーを実行する場合、重要なイベントがモニターされ、ビジュアル・フィードバックが「モニター (*Monitor*)」リスト・ボックスに、'>>> ' 接頭部付きで表示されます。

「サーバー・モニター (Server_Monitor)」モードと「管理モニター (Admin_Monitor)」モードの両方で、これらのイベントを指定されたファイルにログ記録するための追加のオプションが使用可能です。操作可能なソリューションでは、このオプションを使って監

査証跡を提供することも可能です。このオプションをどちらかのモードで開始するには、「ファイル (*File*)」プルダウン・メニューから「ログ (*Log*)」オプションを選択します。このタスクの結果は、ファイル選択ダイアログ・ボックスに表示されます。

ログ・ファイル名 (後続のモニター・イベントが記録される) を選択するには、管理者は「ファイル名 (*File Name*)」入力フィールドに表示される MQSeries Everyplace が生成したログ・ファイル名を受け入れるか (この例では '949679065895_MCSlog')、またはその名前を希望するログ・ファイル名で上書きしてから、「保管 (*Save*)」ボタンをクリックします。

このタスクからの視覚的フィードバックの例は、図47 に示されています。

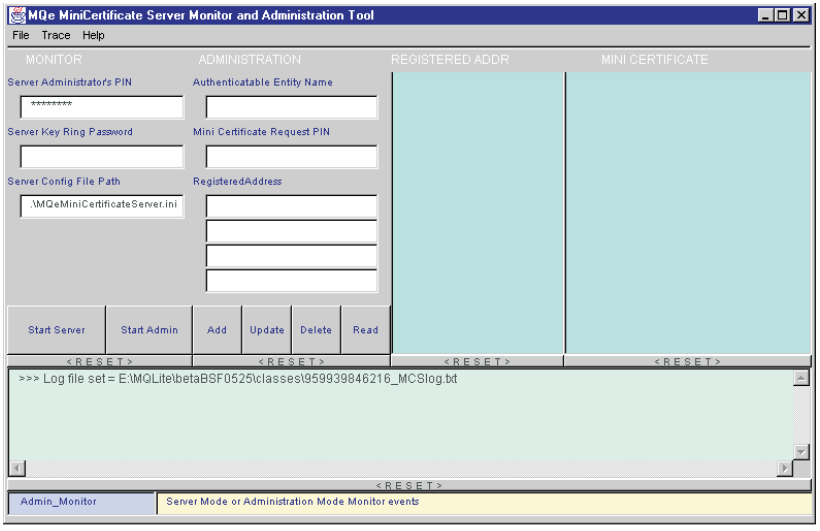

図 47. ミニ認証サーバー・ログ・ファイル名の画面

管理を使って「Bruce」という名前の認証可能なエンティティーを追加するために作成される、ログ・ファイルの例は以下のとおりです。

```
>>> Log file set = E:¥MQLite¥betaBSF0202¥Classes¥949682538438_MCSlog.txt
>>> Admin Mode started OK
>>> opening MiniCertificateServer Registry
>>> MiniCertificateServer Registry opened OK
>>> Entity added OK = Bruce
>>> Entity Registered Address added OK
```

ミニ認証の更新

ミニ認証発行サービスによってエンティティー用に発行された証明書は、発効日から 1 年間有効で、有効期限が切れる前に更新することをお勧めします。更新された証明書

は、同じミニ認証発行サービスから入手されます。更新を要求する前に、最初の認証発行と同じように、要求が発行サービスによって許可されており、一回限りの使用の認証要求 PIN を入手していなければなりません。サーバーを使用して更新用の PIN を入手する場合には、エンティティーを追加しているのではなく、更新しているということを忘れないでください。

エンティティーに対して証明書が発行される際に、ミニ認証サーバーが所有する証明書のコピーが一緒に発行されます。これは、他の証明書の妥当性を検査する際に必要です。 1.2 より前のバージョンの MQSeries Everyplace では、認証サーバーの証明書は、エンティティーの証明書より前に有効期限が切れる可能性があります。このような場合には、エンティティーの証明書の更新を要求して、サーバーの証明書を更新することができます。エンティティーの証明書と共に ミニ認証サーバーの証明書の新しいコピーが戻されます。ミニ認証サーバー バージョン 1.2 から、ミニ認証サーバーの証明書は、エンティティーの証明書より後に有効期限が切れます。

クラス com.ibm.mqe.registry.MQePrivateRegistryConfigure には、更新された証明書を要求するために使用できる **renewCertificates()** というメソッドが含まれています。これは、発行サービスから更新された証明書を要求するコマンド行プログラムをインプリメントするプログラム例 examples.certificates.RenewWTLSCertificates で使用されています。

プログラムには、次の 4 つの必須パラメーターがあります。

```
RenewWTLSCertificates <entity> <ini file> <MCS addr> <MCS Pin>
```

各パラメーターの意味は、次のとおりです。

entity 更新された証明書が要求されるエンティティーの名前。これは、キュー・マネージャー、キュー、またはその他の認証可能なエンティティーのいずれかでなければなりません。キューの名前は、<queue manager>+<queue> の形式 (たとえば、myQM+myQueue) で指定しなければなりません。

ini file レジストリーのセクションが含まれている構成ファイルの名前。これは、一般的には、キュー・マネージャーに使用されている構成ファイルと同じです。キューの場合、一般的には、キューを所有しているキュー・マネージャー用の構成ファイルです。

MCS addr
ミニ認証サーバーのホスト名およびポート・アドレス (たとえば、myServer:8085)

MCS Pin
この更新要求を許可するために ミニ認証サーバーにより発行される一回限り使用する PIN。

新しい信任状の入手 (私用鍵および公開鍵)

証明書を更新すると、既存の公開鍵の更新された証明書を入手します (これによって、既存の私用 / 公開鍵ペアを引き続き使用することができます)。私用 / 公開鍵ペアを変更したい場合には、新しい信任状を要求する必要があります。これには、新しい公開鍵を形成する新しい公開証明書に関するミニ認証発行サービスに対する要求も含まれます。新しい信任状に関する証明書を要求する前に、最初の認証発行と同じように、要求が発行サービスによって許可されており、一回限り使用する認証要求 PIN を入手していなければなりません。 (サーバーを使用して新しい証明書の PIN を入手する場合には、エンティティーを追加しているのではなく、更新しているということを忘れないでください。)

クラス com.ibm.mqe.registry.MQePrivateRegistryConfigure には、新しい信任状を要求するために使用できる **getCredentials()** というメソッドが含まれています。これは、発行サービスから新しい信任状を要求する GUI プログラムをインプリメントするプログラム例 examples.install.GetCredentials で使用されています。

注: 新しい信任状が発行されると、既存の信任状はレジストリーにアーカイブされます。以前の信任状を使用して作成された暗号化メッセージの暗号化を解除することはできなくなります。新しい証明書は、以前の信任状を使用して作成されたディジタル署名 (MQeMTrustAttribute を使用) の妥当性検査を行いません。

ミニ認証のリスト

レジストリー内の証明書をリストすることは、たとえば有効期限日付を調べる場合などに便利です。これは、クラス com.ibm.mqe.attributes.MQeListCertificates のメソッドを使用して行うことができます。これらのメソッドは、証明書をリストするコマンド行プログラムをインプリメントするプログラム例 examples.certificates.ListWTLSCertificates で使用されています。

プログラムには、次の 1 つの必須パラメーターと 3 つのオプショナル・パラメーターがあります。

```
ListWTLSCertificates <reg Name>[<ini file>] [<level>] [<cert names>]
```

各パラメーターの意味は、次のとおりです。

regName

証明書をリストするレジストリーの名前。キュー・マネージャー、キュー、または他のエンティティーに属する私用レジストリーであっても、公開レジストリーであっても、(管理者の場合には) ミニ認証サーバーのレジストリーであっても構いません。キューのレジストリー内の証明書をリストしたい場合には、その名前を <queue manager>+<queue> という形式 (たとえば、myQM+myQueue) で指定する必要があります。公開レジストリー内の証明書をリストしたい場合には、MQeNode_PublicRegistry という名前でなければなりません。これ以外の名前では、公開レジストリーとして機能しません。ミニ認証サーバーのレジストリーの名前は、MiniCertificateServer です。

ini file レジストリーのセクションが含まれている構成ファイルの名前。これは、一般的には、キュー・マネージャーまたはミニ認証サーバーに使用されている構成ファイルと同じです。キューの場合、一般的には、キューを所有しているキュー・マネージャー用の構成ファイルです。このパラメーターは、パラメーターを省略できる公開レジストリー以外のすべてのレジストリーに対して指定しなければなりません。

level リストの詳細度のレベル。これは、次のいずれかになります。

-b または -brief
証明書の名前を 1 行に 1 つずつ印刷する

-n または -normal
証明書の名前を 1 行に 1 つずつ、タイプ (古いフォーマットか新しいフォーマット) の前に印刷する

-f または -full 証明書の名前、タイプ、および内容の一部を印刷する

このパラメーターはオプションで、省略した場合には、″normal″ レベルの詳細度が使用されます。

cert names
リストする証明書の名前のリスト。フラグ `-cn` で始まり、その後に証明書の名前が続きます。たとえば、`-cn ExampleQM putQM` のようになります。このパラメーターが使用されると、指定した証明書だけがリストされます。このパラメーターを省略すると、レジストリー内のすべての証明書がリストされます。

MQSeries Everyplace V1.2 で更新されたミニ認証フォーマット

MQSeries Everyplace によって使用されるミニ認証は、WAP によって使用される WTLS 証明書に基づいています。 MQSeries Everyplace バージョン 1.0 および 1.1 で使用されていた証明書は、開発時に使用可能であった WTLS 仕様の最新ドラフトに基づいていました。それ以降、証明書の標準が承認されました。MQSeries Everyplace V1.2 では、承認されている標準に準拠した新しいミニ認証が取り入れられています。

MQSeries Everyplace V1.2 は、以前のフォーマットのミニ認証と更新されたフォーマットのミニ認証の両方をサポートします。よって、以前のフォーマットのミニ認証を使用している場合にも、引き続きそれを使用することができます。ただし、できるだけ早く、新しい証明書に移行することをお勧めします。以前のフォーマットに対するサポートは、MQSeries Everyplace バージョン 2 で終了する予定です。

MQSeries Everyplace V1.2 から ミニ認証サーバーを実行して証明書を更新することによって、証明書を新しいフォーマットにアップグレードできます (272ページの『ミニ認証の更新』 を参照してください)。更新された証明書は、新しいフォーマットになります。

互換モード

デフォルトでは、MQSeries Everyplace V1.2 のミニ認証サーバーは、証明書を新しいフォーマットで発行します。新しいフォーマットの証明書は、以前のバージョンの MQSeries Everyplace では使用できません。証明書をすでに使用しており、関係のあるすべてのキュー・マネージャー・ソフトウェアをV1.2 にアップグレードしていない場合には、すべてのソフトウェアがアップグレードされるまで、オリジナル・フォーマットの証明書を引き続き使用したいことがあります。ミニ認証サーバーを*互換モード* で実行されるように構成して、オリジナル・フォーマットの証明書を発行することができます。このようにするには、新しいセクション [Mode] を構成ファイルに追加します。このセクションには、以下の例のように、Mode=old というエントリーが 1 つ含まれていなければなりません。

オリジナル・フォーマットの証明書を発行するための構成ファイルの例

```
[Alias]
*
*        Event log class
*
(ascii)EventLog=examples.log.LogToDiskFile
*
*        Network adapter class
*
(ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
*
*        Queue Manager class
*
(ascii)QueueManager=com.ibm.mqe.MQeQueueManager
*
*        Trace handler (if any)
*
(ascii)Trace=examples.awt.AwtMQeTrace
*
*        Message Log file interface
*
(ascii)MsgLog=com.ibm.mqe.adapters.MQeDiskFieldsAdapter
*
*    Mini Certificate Server Registry class
*
(ascii)MiniCertSvrRegistry=com.ibm.mqe.registry.MQeMiniCertSvrRegistry
*
*    Mini Certificate Server Issuance Manager class
*
(ascii)MiniCertIssuanceManager=com.ibm.mqe.server.MQeMiniCertIssuanceManager
*------------------------------------------------------------------------*
[ChannelManager]
*
*       Maximum number of channels allowed
*
(int)MaxChannels=0
*------------------------------------------------------------------------*
[Listener]
*
*       FileDescriptor for listening adapter
*
(ascii)Listen=Network::8085
*
*       FileDescriptor for Network read/write
*
(ascii)Network=Network:
*
*       Channel timeout interval in seconds
*
```

```
(int)TimeInterval=300
*
*    Mini Certificate Server Registry class
*
*--------------------------------------------------------------------------*
[MiniCertSvrRegistry]
*
*   Mini-Certificate-Server Registry's Root User InitialPIN
*
(ascii)InitialPIN=12345678
*
*   Mini Certificate Server Registry's KeyRingPassword
*
(ascii)KeyRingPassword=It_is_a_secret
*--------------------------------------------------------------------------*
[Mode]
*
*   Issue certificates in the old format
*
(ascii)Mode=old
*
```

関係のあるすべてのキュー・マネージャー・ソフトウェアが MQSeries Everyplace V1.2 にアップグレードされたら、 `Mode=old` を `Mode=new` に変更するか、`[Mode]` セクションを省略して、新しいフォーマットの証明書の発行を開始できます。

注: 新しいミニ認証発行サービスをセットアップする場合、(ミニ認証サーバーを初めて実行する場合)、サーバー自体は、新しい信任状および新しい証明書を発行します。サーバーの所有する証明書は、互換性モードで稼働している場合でも、常に新しいフォーマットです。新しいサービスをセットアップする場合には、互換性モードを使用しないでください。

第9章 MQSeries Everyplace でのトレース

このセクションでは、 MQSeries Everyplace トレース・プログラムの使用とカスタマイズを助ける情報を提供しています。

MQSeries Everyplace は、単純で、かつ役立つトレース機能を提供します。この機能は、プログラムの実行中、または後でファイルに記録される実行の追跡を検査することによって、プログラムの実行の経路をたどるのに使用することができます。トレース・メッセージは実行中のコードから、トレース・ウィンドウに送信され、メッセージはそこに表示されます。

トレース機能はトレース専門であり、中断ポイントを設定して解放する機能など、デバッガーに備えられている一部の機能は含まれていません。

トレース・クラスの例は、 examples.trace サブディレクトリーにあります。これらのクラスの詳細については、 28ページの『examples.trace パッケージ』を参照してください。これらのクラスは、実行中の MQSeries Everyplace 環境からトレースを処理し、表示する際に使用できます。281ページの『MQeTrace のサンプル』で、サンプル・ファイルの使用方法について説明します。

トレースは通常、問題の診断を除いて、実稼働環境で使用されることはありません。どの形式であっても、トレースは MQSeries Everyplace のパフォーマンスに影響するためです。

トレースの使用

アプリケーション・プログラムの実行をトレースするには、次の例が示すように、MQe.trace メソッドを使ってコードの適切な場所にステートメントを入れる必要があります。

```
...
/* */
trace( "We got here" );
...
```

実行時、この結果テキスト ″We got here″ が MQSeries Everyplace トレース・ウィンドウに表示されます。

トレース・メッセージ・フォーマット

メッセージにはいくつかのタイプ (通知、警告、エラー、セキュリティー、およびデバッグ) があり、タイプは 280ページの表12 で説明されているように、最初の文字で示されます。

表 12. トレース・メッセージのタイプ

最初の文字	意味
I または i	通知
W または w	警告
E または e	エラー
S または s	セキュリティー
D または d	デバッグ

大文字の接頭部はアプリケーション・トレース・メッセージに使用され、小文字の接頭部はシステム・トレース・メッセージに使用されます。システム・トレース・メッセージは、通常 MQSeries Everyplace の内部からのみ生成されます。

メッセージは、メッセージのレベルを検査する MQSeries Everyplace トレース機能に送信され、必要であればそれをトレース・ウィンドウに出力します。認識可能な接頭部を持つトレース・メッセージは System.err に書き込まれ、その他のメッセージは System.out に書き込まれます。

examples.trace ディレクトリーの examples.trace.MQeTrace ファイルには、 MQSeries Everyplace 内部ルーチンが発行するメッセージのさまざまなメッセージ・テンプレートが含まれます。メッセージの形式は次のとおりです。

```
/* common messages */
    {    "1",            "d;[00001]:Created"                    },
    {    "2",            "d;[00002]:Destroyed"                  },
    {    "3",            "d:[00003]:Close"                      },
    {    "4",            "w:[00004]:Warning:#0"                 },
    {    "5",            "e:[00005]:Error:#0"                   },
    {    "6",            "i:[00006]:Command:#0"                 },
    {    "7",            "i:[00007]:Waiting"                    },
    {    "8",            "i:[00008]:#0 input byte count=#1"     },
    ...,
```

ここで、先頭のストリングはメッセージ番号で、 2 番目のストリングはメッセージ・テンプレートです。

examples.trace.MQeTraceResource には、メッセージ・ストリングが英語で入っています。その他の言語のバージョンも、このディレクトリーで提供されます。

テンプレートの形式は、次のとおりです。

- 表12 で説明されるメッセージ・タイプ
- 修飾子文字。この修飾子には次の意味があります。

表 13. トレース・メッセージの修飾子

修飾子	意味
:	変更は適用されない
;	作成 / 破棄オブジェクトのために予約済み

表 13. トレース・メッセージの修飾子 (続き)

修飾子	意味
+	ログ・インターフェースを介してこのメッセージをログに記録する
¬	無視 - このメッセージは表示しない

- 形式 '[nnnnn]:' のメッセージ番号
- メッセージ・テキスト。これには '#n' という書式が挿入されています ('n' は 0 〜 9 の整数)。

このソース・ファイルを修正することによって、メッセージの種別を変更することができます。たとえば、警告からエラーに変更したり、修飾子文字を ':' から '+' に変更することによって、メッセージをイベント・ログにコピーしたりできます。

新しいトレース・メッセージは、addMessage または addMessageBundle 呼び出しを使って、実行時に追加できます。たとえば、1 つの新規メッセージを追加する方法は次のとおりです。

```
...
MQeTraceInterface MyTrace = MQe.GetTraceHandler();
myTrace.addMessage(" :[11111]:My Application - #0 = #1" );
...
trace( 11111, new String[] { "Magic word", "xyzzy" } );
...
```

トレースの活動化

デフォルトでは活動状態でないトレースは、以下のコードで示すように、MQe.setTraceHandler を使って活動化できます。

```
...
  /* give the trace object to MQe */
  setTraceHandler( new myTraceHandler() );
  trace( "I:Starting..." );
  ...
```

MQSeries Everyplace ツールキットの一部として出荷されるトレース・ハンドラーのサンプルには、トレース活動化コードが含まれます。

トレースのカスタマイズ

サンプル・ディレクトリーで提供されるトレース・クラスは、カスタム・トレース・ハンドラーとして使用できます。

MQeTrace のサンプル

MQeTrace のサンプル・クラスは、デフォルトではトレース・メッセージを System.out および System.err、またはそのどちらかに出力する、単純なトレース機能を提供します。

トレース・ウィンドウを活動化するには、次のコードを指定します。

```
...
/* Start the example version of MQeTrace  */
new examples.trace.MQeTrace( "Trace", null ) );
...
trace( "I:Starting..." );
trace( 123456, "Insert" );
...
```

コンストラクターの 2 番目のパラメーターはトレース・メッセージに使用される言語で、ヌルが指定されると、デフォルト言語が使用されます。別の方法として、以下のように、メッセージの種別を変更する、異なるリソース・ファイルを指定することもできます。

```
...
/* Start the example version of MQeTrace  */
new examples.trace.MQeTrace( "Trace", "MyMessageResourceFile") );
...
trace( "I:Starting..." );
trace( 123456, "Insert" );
...
```

現在活動状態のトレース・ハンドラー・オブジェクトは、 MQe.getTraceHandler メソッド呼び出しを発行することによって見つけることができます。この参照を使用して、トレースの動作 (書き込まれるトレースのタイプを選択または選択解除する) を変更できます。

```
...
  /* Start the example version of MQeTrace  */
  MQeTraceInterface trace = MQe.getTraceHandler( );
  if ( trace instanceof MQeTrace )
    {
    ((MQeTrace) trace).MsgInf   = true;
    ((MQeTrace) trace).MsgDebug = true;
    ((MQeTrace) trace).MsgTime  = true;
    }
  ...
  trace( "I:Starting..." );
  ...
```

変更できる MQeTrace の変数 (およびそのデフォルト) は次のとおりです。

```
  public   boolean   MsgInf       = false;      /* Informaton msgs      */
  public   boolean   MsgWarn      = true;       /* warning msgs         */
  public   boolean   MsgErr       = true;       /* error msgs           */
  public   boolean   MsgSecurity  = false;      /* Security  msgs       */
  public   boolean   MsgSys       = true;       /* System modifier      */
  public   boolean   MsgDebug     = false;      /* Debug modifier       */
  public   boolean   MsgLog       = false;      /* Trace message to log */
  public   boolean   MsgTime      = false;      /* add Time stamp       */
  public   boolean   MsgPrefix    = false;      /* add object prefix    */
  public   boolean   MsgThread    = false;      /* add Thread ID        */
```

詳細は、examples.trace ディレクトリーで、 MQeTrace のソース・コードを調べることによって参照できます。

このトレースのサンプルを、より洗練されたトレース・プログラムの基礎として使用することもできますし、全く新しいトレースを作成することもできます。

アプリケーション・プログラムは、MQeTraceInterface をインプリメントし、MQe.setTraceHandler メソッド呼び出しを発行するだけで、通常の機能だけでなくトレース・ハンドラーにもなることができます。

トレース用のグラフィカル・ユーザー・インターフェース

examples.trace ディレクトリーで提供される基本トレース機能は、アプリケーションと関連するコンソール・ウィンドウに、 System.out および System.err に関するトレース・メッセージを表示するだけです。

examples.awt ディレクトリーには、Java AWT のサブセットを使用してグラフィカル・ユーザー・インターフェースをトレースに提供する別のトレース・ハンドラーもあります。これによって、さまざまなトレース・オプションを動的に変更することができます。

```
...
/* Start the example GUI version of MQeTrace */
new examples.awt.AwtMQeTrace( "My Trace title", null ) );
...
trace( "I:Starting..." );
```

このコードは、タイトル 'My Trace' でトレース・ウィンドウを開始し、通知メッセージ "I:Starting" を表示します。トレース・ウィンドウには、ユーザーがトレースのレベル、メッセージのフォーマット、および他のプロパティーを変更するためのプルダウン・メニューがあります (284ページの図48 に示されています)。トレースの実行には MQSeries Everyplace オブジェクトが必要であることに注意してください。上記の例では、コードが基本 MQSeries Everyplace クラスを拡張するクラスの一部であることが前提になっていました。それ自体は MQSeries Everyplace を拡張しないオブジェクトから、 MQSeries Everyplace トレース・メッセージを出力することが可能です。この場合、MQSeries Everyplace オブジェクトを作成してから、このオブジェクトのメソッドを使って、トレースを指定することが必要です。たとえば、次のようにします。

```
...
/* create a MQe object */
MQe dbg = new MQe( );
dbg.Message( "D:We got here" );
...
```

MQSeries Everyplace トレースは、現行の Java 仮想計算機のスレッド上で実行される MQSeries Everyplace オブジェクトからのすべてのメッセージが、同じトレース機能によって処理され、同じトレース・ウィンドウに表示されるように、Java 仮想計算機全体で

実行されます。これは、イベントが実際に発生する順番を示すので、大きな利点となり得ます。しかし、異なるスレッド上で発生する完全に独立したイベントを分離したい場合は、欠点になることもあります。

注: MQSeries Everyplace トレース・ウィンドウを終了しても、 Java プログラムは終了しません。

AWT トレース・ウィンドウのレイアウト例

トレースに関連するグラフィカル・ユーザー・インターフェース・コンポーネントで使用されるテキストを指定する、`MyMessageResourceFileGUI` ファイルが必要であることに注意してください。

`examples.awt` 中のトレース・プログラムの例は、 図48 で示されるレイアウトでウィンドウを表示します。

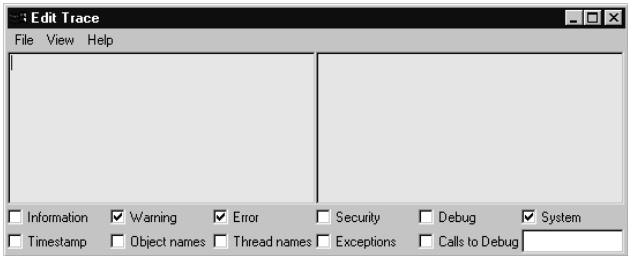

図 48. トレース GUI ウィンドウのサンプル

メニュー項目は次のとおりです。

- 「ファイル (File)」メニュー

  消去 (Clear)
  トレース・ウィンドウを消去します。

  別名保管... (Save As...)
  トレース・ウィンドウの内容をディスク・ファイルに保管します。

  ログへのトレース (Trace to Log)
  トレース・メッセージをイベント・ログにコピーします。

  トラップ入出力 (Trap I/O)
  System.out および System.err への出力がウィンドウに表示されます。このオプションをチェックしないと、出力は Java コンソールに移動します。

強制終了 (Kill)
トレースと所有するアプリケーションの両方を終了します。ウィンドウ・フレームの「終了 (Exit)」ボタンをクリックすると、トレースだけが終了します。

- 「表示 (View)」メニュー

表示オプション (View Options)
トレース・メッセージ表示オプションを表示します。

System.out
「System.out」ウィンドウを表示します。

System.err
「System.err」ウィンドウを表示します。

さまざまなトレース・メッセージの表示オプションは、*System.err.println* ウィンドウにトレース・メッセージが表示される方法、および表示されるトレース・メッセージを制御します。

通知 (Information)
通知メッセージを表示します。

警告 (Warning)
警告メッセージを表示します。

エラー (Error)
エラー・メッセージを表示します。

セキュリティー (Security)
セキュリティー・メッセージを表示します。

デバッグ (Debug)
デバッグ通知メッセージを表示します。

システム (System)
システム特性を持つメッセージが表示されるかどうかを制御します。これは、通知 (Information)、警告 (Warning)、エラー (Error)、セキュリティー (Security)、およびデバッグ (Debug) スタイルのメッセージに影響を与えます。

タイム・スタンプ (Timestamp)
メッセージの先頭に現在のタイム・スタンプを付けます。

オブジェクト名 (Object names)
メッセージの先頭にオブジェクト・タイプ、およびメッセージから発生するインスタンスを付けます。

スレッド名 (Thread names)
メッセージの先頭に、現在実行中のスレッド名を付けます。

例外 (Exceptions)
MQeException が出されるときにはスタック・トレースを表示します。

デバッグへの呼び出し (Calls to Debug)
アプリケーションまたは MQSeries Everyplace が MQe.Debug 呼び出しを発行する時にスタック・トレースを表示します。

'System.err.println and Trace message filter' は、出力内のものを一致させるのに使用されるストリングです。一致すると出力が表示され、一致しないと出力は表示されません。

この機能を使用して、特定のスレッドから選択的にメッセージを表示できます (「スレッド名 (Thread name)」チェック・ボックスがチェックされていることが前提となっています)。

トレース・オプションの設定

検査可能なコンポーネントのいずれかを事前検査する新しい GUI リソース・ファイルを作成することによって、AwtMQeTrace プログラムの始動時にさまざまなトレース・オプションを事前設定できます。たとえば次のとおりです。

```
public class MQeTraceResourceGUI extends java.util.ListResourceBundle
  {
  static final Object[][] contents = {
  /*    Check items can be pre-checked by replacing the blank with an "!"  */
           { "File",            "File"           },
           {   "Clear",         "Clear"          },
           {   "Save",          "Save As..."     },
           {   "Log",           " Trace to Log"  },    /* check item      */
           {   "Trap",          "!Trap I/O"      },    /* check item      */
           {   "Halt",          "Kill"           },
           { "View",            "View"           },
           {   "Options",       "!View Options"  },    /* check item      */
           {   "SystemOut",     "!System.out"    },    /* check item      */
           {   "SystemErr",     " System.err"    },    /* check item      */
           { "Help",            "Help"           },
           {   "About",         "About..."       },
  /* checkbox labels                                                      */
           { "Information",     " Information"   },    /* check item      */
           { "Warning",         "!Warning"       },    /* check item      */
           { "Error",           "!Error"         },    /* check item      */
           { "Debug",           " Debug"         },    /* check item      */
           { "Security",        " Security"      },    /* check item      */
           { "System",          "!System"        },    /* check item      */
           { "Timestamp",       " Timestamp"     },    /* check item      */
           { "Objects",         " Object names"  },    /* check item      */
           { "Threads",         " Thread names"  },    /* check item      */
           { "Exceptions",      " Exceptions"    },    /* check item      */
           { "CallStack",       " Calls to Debug" },   /* check item      */
  /* About dialog                                                         */
   { "AboutTitle",      "About MQe Trace" },
   { "AboutVersion",    "MQe version"     },
   { "AboutProduct",    "Product number 5639-I47"     },
   { "AboutCopyright",  "(C) Copyright IBM Corp. 1999 All Rights Reserved" },
```

```
  { "AboutCopyright2", "Licensed Materials - Property of IBM" },
  { "AboutTrace",      "Trace version"   },
  { "AboutComments",   " "               },
  { "OK",              "OK"              },
                                  };
public Object[][] getContents( )
  {
  return( contents );
  }
```

注: トレース・オプションが MQeTrace で、次のコードで示されているように方針に基づいて変更される場合、「AwtMQeTrace」ウィンドウにある対応するコンポーネントは更新されません。

```
  ...
  /* Start the example version of MQeTrace  */
  MQeTraceInterface trace = MQe.getTraceHandler( );
  if ( trace instanceof MQeTrace )
    ((MQeTrace) trace).MsgDebug = true;
  ...
```

第10章 MQSeries Everyplace アダプター

MQSeries Everyplace アダプターは、MQSeries Everyplace を装置接続機構にマップする際に使用します。アダプターは、通常、伝送プロトコルに関連していますが、ストレージ・デバイスなどの他のインターフェースと使用することもできます。

以下のアダプターが、MQSeries Everyplace で提供されています。詳細は、「*MQSeries Everyplace for Multiplatforms プログラミング・ガイド*」を参照してください。

MQeDiskFieldsAdapter
ローカル・ディスクに対する MQeFields 情報の読み取りおよび書き込みに関するサポートを提供します。

MQeMemoryFieldsAdapter
MQeFields 情報に関する非永続ストアを提供します。

MQeReducedDiskFieldsAdapter
ローカル・ディスクに対する MQeFields 情報の高速書き込みに関するサポートを提供します。

MQeTcpipAdapter
TCP/IP ストリーム上のデータの読み取りおよび書き込みに関するサポートを提供します。

MQeTcpipHttpAdapter
HTTP 1.0 プロトコル用の基本サポートを提供するように、MQeTcpipAdapter を拡張します。

MQeTcpipLengthAdapter
単純なバイト効率プロトコルを提供するように、MQeTcpipAdapter を拡張します。

MQeTcpipHistoryAdapter
最新に使用されたデータをキャッシュに入れる、より効率的なプロトコルを提供するように、 MQeTcpipLengthAdapter を拡張します。

MQeUdpipAdapter
UDP/IP データグラム上の保証されたデータ転送に関するサポートを提供します。

MQeWESAuthenticationAdapter
Websphere Everyplace 認証プロキシーおよび透過プロキシーを介する HTTP 要求のトネリングに関するサポートを提供します。

独自のアダプターを作成して、MQSeries Everyplace をユーザー固有の環境に調整することもできます。次のセクションでは、このような作業に役立つアダプターの例をいくつか紹介します。

アダプターの例

このセクションでは、MQSeries Everyplace アダプターの作成について説明します。 次のアダプターについて説明します。

簡単な通信アダプター
この例では、標準の Java クラスを使用して TCPIP を操作し、特別のプロトコルを追加します。

簡単なメッセージ・ストア・アダプター
この例では、メッセージ・ストア用のインターフェースとして使用されるアダプターを作成します。

MQSeries Everyplace で提供されているアダプター・クラスの詳細については、MQSeries Everyplace for Multiplatforms プログラミング・リファレンスの第 9 章を参照してください。

簡単な通信アダプターの例

この例では、標準の Java クラスを使用して TCPIP を操作し、その先頭に独自のプロトコルを追加します。このプロトコルには、実データの後ろにあるデータ・パケットに 4 バイト長のデータから成るヘッダーがあります。このヘッダーよって、着信側には予想されるデータ量が知らされます。

この例は、MQSeries Everyplace で提供されるアダプターと置き換えることを意図したものではなく、むしろ通信アダプターの作成方法を簡単に紹介しているに過ぎません。実際のところ、エラーの処理、リカバリー、およびパラメーターの検査においては、もっと多くの注意を払う必要があります。使用する MQSeries Everyplace 構成によっては、提供されているアダプターで十分な場合もあります。

MQeAdapter の属性を継承して、新しいクラス・ファイルが構成されます。一部の変数は、このアダプターのインスタンス情報、すなわちホスト名、ポート番号、および出力ストリーム・オブジェクトを保持するように定義されています。

オブジェクトには MQeAdapter のコンストラクターが使用されるため、コンストラクターに付加的なコードを追加する必要はありません。

```
public class MyTcpipAdapter extends MQeAdapter
  {
  protected     String               host         = "";
  protected     int                  port         = 80;
  protected     Object               readLock     = new Object( );
  protected     ServerSocket         serversocket = null;
  protected     Socket               socket       = null;
  protected     BufferedInputStream  stream_in    = null;
  protected     BufferedOutputStream stream_out   = null;
  protected     Object               writeLock    = new Object( );
```

次に、activate メソッドがコード化されます。これは、ファイル記述子から、ターゲット・ネットワーク・アドレスの名前 (コネクターの場合) や聴取ポート (リスナーの場合) を取り出すメソッドです。 fileDesc パラメーターには、アダプターのクラス名または別名、およびそのアダプターに関するすべてのネットワーク・アドレス・データ (例、MyTcpipAdapter:127.0.0.1:80) が含まれます。 thisParam パラメーターには、管理によって接続が定義される際に設定されたすべてのデータが含まれます。なお、このパラメーターの値は、通常 "?Channel" のようになります。 thisOpt パラメーターには、管理によって設定されたアダプターのセットアップ・オプションが含まれます。たとえば、このアダプターが着信の接続に対して聴取を行うものである場合は、MQe_Adapter_LISTEN が含まれます。

```
public void   activate( String        fileDesc,
                        Object        thisParam,
                        Object        thisOpt,
                        int           thisValue1,
                        int           thisValue2 ) throws Exception
  {
  super.activate( fileDesc,
                  thisParam,
                  thisOpt,
                  thisValue1,
                  thisValue2 );
  /* isolate the TCP/IP address - "MyTcpipAdapter:127.0.0.1:80"     */
  host = fileId.substring( fileId.indexOf( ':' ) + 1 );
  i    = host.indexOf( ':' );                /* find delimiter     */
  if ( i > -1 )                              /* find it ?          */
    {
    port = (new Integer( host.substring( i + 1 ) )).intValue( );
    host = host.substring( 0, i );
    }
  }
```

出力ストリームをクローズし、すべての残りのデータをストリーム・バッファーからフラッシュするためには、 close メソッドを定義する必要があります。 close は、クライアント / サーバー間のセッションで何回も呼び出されますが、チャネルが完全にアダプターの使用を終えると、今度は close がオプション MQe_Adapter_FINAL を指定して MQSeries Everyplace を呼び出します。チャネルの存続期間ごとに 1 つのソケット接続を持つアダプターでは、オプション MQe_Adapter_FINAL を指定して呼び出しを設定し、ソケットを実際にクローズするために使用されるアダプターでは、バッファーをフラッシュするだけの別の呼び出しを行います。ただし、各要求ごとに新しいソケットが使用される場合は、MQSeries Everyplace への各呼び出しごとにソケットがクローズされ、次の open 呼び出しでまた新しいソケットが割り振られます。

```
public void    close(  Object  opt ) throws Exception
  {
  if ( stream_out   != null )                /* output stream ?    */
    {
    stream_out.flush();                      /* empty the  buffers */
    stream_out.close();                      /* close it           */
    stream_out = null;                       /* clear              */
```

```
    }
  if ( stream_in    != null )                /* input stream ?      */
    {
    stream_in.close();                       /* close it            */
    stream_in = null;                        /* clear               */
    }
  if ( socket       != null )                /* socket ?            */
    {
    socket.close();                          /* close it            */
    socket = null;                           /* clear               */
    }
  if ( serversocket != null )                /* serversocket ?      */
    {
    serversocket.close();                    /* close it            */
    serversocket = null;                     /* clear               */
    }
  host = "";
  port = 80;
  }
```

MQe_Adapter_ACCEPT 要求を処理して接続要求の着信を受け入れるためには、 control メソッドをコード化する必要があります。これは、ソケットがリスナー (サーバー・ソケット) である場合にのみ可能です。聴取ソケットに指定されたすべてのオプション (MQe_Adapter_LISTEN を除く) は、受け入れの結果として作成されたソケットにコピーされます。これを行うためには、MQe_Adapter_SETSOCKET という別の制御オプションを使用します。このオプションを使用して、インスタンス化されたばかりのアダプターにソケット・オブジェクトを渡すことができます。

```
public Object control( Object opt, Object ctrlObj ) throws Exception
  {
  if ( checkOption( opt, MQe.MQe_Adapter_LISTEN      ) &&
       checkOption( opt, MQe.MQe_Adapter_ACCEPT ) )
    {
    /* CtrlObj - is a string representing the file descriptor of the */
    /*           MQeAdapter object to be returned e.g. "MyTcpip:"    */
    Socket  ClientSocket = serversocket.accept();   /* wait connect  */
    String  Destination  = (String) ctrlObj;        /* re-type object*/
    int i  = Destination.indexOf( ':' );
    if ( i < 0 )
      throw new MQeException( MQe.Except_Syntax,
                              "Syntax:" + Destination );
    /* remove the Listen option                                      */
    String NewOpt = (String) options;         /* re-type to string   */
    int j  = NewOpt.indexOf( MQe.MQe_Adapter_LISTEN );
    NewOpt = NewOpt.substring( 0, j ) +
             NewOpt.substring( j + MQe.MQe_Adapter_LISTEN.length( ) );
    MQeAdapter Adapter = MQe.newAdapter( Destination.substring( 0,i+1 ),
                                         parameter,
                                         NewOpt + MQe_Adapter_ACCEPT,
                                         -1,
                                         -1 );
    /* assign the new socket to this new adapater                    */
    Adapter.control( MQe.MQe_Adapter_SETSOCKET, ClientSocket );
```

```
      return( Adapter );
      }
    else
    if ( checkOption( opt, MQe.MQe_Adapter_SETSOCKET ) )
      {
      if ( stream_out != null )  stream_out.close();
      if ( stream_in  != null )  stream_in .close();
      if ( ctrlObj    != null )                 /* socket supplied ?   */
        {
        socket     = (Socket) ctrlObj;          /* save the socket     */
        stream_in  = new BufferedInputStream ( socket.getInputStream () );
        stream_out = new BufferedOutputStream( socket.getOutputStream() );
        }
    else
      return( super.control( opt, ctrlObj ) );
  }
```

open メソッドでは、聴取のソケットやコネクターのソケットを検査し、適当なソケット・オブジェクトを作成する必要があります。入力および出力ストリームの再初期設定を行うためには、 control メソッドを使用し、これに新しいソケット・オブジェクトを渡します。 opt パラメーターは、MQe_Adapter_RESET に設定することができます。これは、直前の操作がすべて完了し、新しい任意の読み取りまたは書き込みによって新しい要求を構成することを意味します。

```
  public void  open( Object opt ) throws Exception
    {
    if ( checkOption( MQe.MQe_Adapter_LISTEN ) )
      serversocket = new ServerSocket( port, 32 );
    else
      control( MQe.MQe_Adapter_SETSOCKET, new Socket( host, port ) );
    }
```

read メソッドでは、読み取り可能な最大レコード・サイズを指定するパラメーターを使用することができます。

この例では、内部ルーチンを呼び出してデータ・バイトの読み取りとエラー・リカバリー (それが適当な場合) を行い、次いで、読み取られる数バイトのデータについて正確な長さのバイト配列を返します。このソケットで 1 度に複数の読み取りが行われることのないよう、注意を払う必要があります。 opt パラメーターには、次の値を設定することができます。

MQe_Adapter_CONTENT
すべてのメッセージの内容を読み取る

MQe_Adapter_HEADER
すべてのヘッダー情報を読み取る

```
{ public byte[] read( Object opt, int recordSize ) throws Exception
    int Count = 0;                            /* number bytes read   */
    synchronized ( readLock )                 /* only one at a time  */
      {
      if ( checkOption( opt, MQe.MQe_Adapter_HEADER    ) )
```

```
        {
        byte lreclBytes[] = new byte[4];        /* for the data length */
        readBytes( lreclBytes, 0, 4 );          /* read the length     */
        int  recordSize = byteToInt( lreclBytes, 0, 4 );
        }
      if ( checkOption( opt, MQe.MQe_Adapter_CONTENT   ) )
        {
        byte Temp[] = new byte[recordSize];     /* allocate work array */
        Count = readBytes( Temp, 0, recordSize);/* read data           */
        }
      }
    if ( Count < Temp.length )                  /* read all length ?   */
      Temp    = MQe.sliceByteArray( Temp, 0, Count );
    return ( Temp );                            /* Return the data     */
    }
```

readByte メソッドは、ソケットから単一バイトのデータを読み取り、エラーがあった場合は指定された回数の再試行を試みたり、あるいは、読み取れる以上のデータが存在する場合はファイルの終わり例外を出すよう設計された内部ルーチンです。

```
  protected int readByte( ) throws Exception
    {
    int intChar    = -1;                        /* input characater    */
    int RetryValue =  3;                        /* error retry count   */
    int Retry = RetryValue + 1;                 /* reset retry count   */
    do{                                         /* possible retry      */
      try                                       /* catch io errors     */
        {
        intChar = stream_in.read();             /* read a character    */
        Retry   = 0;                            /* dont retry          */
        }
      catch ( IOException e )                   /* IO error occured    */
        {
        Retry   = Retry - 1;                    /* decrement           */
        if ( Retry == 0 )  throw e;             /* more attempts ?     */
        }
      } while ( Retry != 0 );                   /* more attempts ?     */
    if ( intChar == -1 )                        /* end of file ?       */
      throw new EOFException();                 /* ... yes, EOF        */
    return( intChar );                          /* return the byte     */
    }
```

readBytes メソッドは、ソケットから数バイトのデータを読み取り、エラーがあった場合は指定された回数の再試行を試みたり、あるいは、読み取れる以上のデータが存在する場合はファイルの終わり例外を出すよう設計された内部ルーチンです。

```
  protected int readBytes( byte buffer[], int offset, int recordSize )
    throws Exception
    {
    int RetryValue = 3;
    int i = 0;                                  /* start index         */
    while ( i < recordSize )                    /* got it all in yet ? */
      {                                         /* ... no              */
      int NumBytes = 0;                         /* read count          */
```

```
  /* retry any errors based on the QoS Retry value                */
  int Retry = RetryValue + 1;              /* error retry count   */
  do{                                      /* possible retry      */
    try                                    /* catch io errors     */
      {
      NumBytes = stream_in.read( buffer, offset + i, recordSize - i );
      Retry    = 0;                        /* no retry            */
      }
    catch ( IOException e )                /* IO error occured    */
      {
      Retry    = Retry - 1;                /* decrement           */
      if ( Retry == 0 )  throw e;          /* more attempts ?     */
      }
    } while ( Retry != 0 );                /* more attempts ?     */
  /* check for possible end of file                               */
  if ( NumBytes < 0 )                      /* errors ?            */
    throw new EOFException( );             /* ... yes             */
  i = i + NumBytes;                        /* accumulate          */
  }    return ( i );                       /* Return the count    */
}
```

readln メソッドは、0x0A の文字で終わるバイト・ストリングを読み取ります。このメソッドでは 0x0D 文字は無視されます。

```
{
synchronized ( readLock )                /* only one at a time  */
  {
  /* ignore the 4 byte length                                     */
  byte lreclBytes[] = new byte[4];         /* for the data length */
  readBytes( lreclBytes, 0, 4 );           /* read the length     */
  int intChar    = -1;                     /* input characater    */
  StringBuffer Result = new StringBuffer( 256 );
  /* read Header from input stream                                */
  while ( true )                           /* until "newline"     */
    {
    intChar = readByte( );                 /* read a single byte  */
    switch ( intChar )                     /* what character      */
      {                                    /*                     */
      case -1:                             /* ... no character    */
        throw new EOFException();          /* ... yes, EOF        */
      case 10:                             /* eod of line         */
        return( Result.toString() );       /* all done            */
      case 13:                             /* ignore              */
        break;
      default:                             /* real data           */
        Result.append( (char) intChar );   /* append to string    */
      }                                    /* end of line ?       */
    }
  }
}
```

status メソッドは、アダプターに関する状況情報を返します。この例では、オプション MQe_Adapter_NETWORK に対してネットワークのタイプ (TCPIP) が返され、オプション MQe_Adapter_LOCALHOST に対して tcpip のローカル・ホスト・アドレスが返されます。

```
public String status( Object  opt ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_NETWORK ) )
      return( "TCPIP" );
    else
    if ( checkOption( opt, MQe.MQe_Adapter_LOCALHOST ) )
      return( InetAddress.getLocalHost( ).toString() );
    else
    return( super.status( opt ) );
    }
```

write メソッドは、データのブロックをソケットに書き込みます。ソケットに対する書き込みは、1 度に 1 つしか実行できないことを確認してください。この例で、このメソッドは内部ルーチン writeBytes を呼び出して実データを書き込み、任意の適当なエラー・リカバリーを実行します。

opt パラメーターには、次の値を設定することができます。

MQe_Adapter_FLUSH
バッファー内のすべてのデータをフラッシュする

MQe_Adapter_HEADER
すべてのヘッダー・レコードを書き込む

MQe_Adapter_HEADERRSP
すべてのヘッダー応答レコードを書き込む

```
  public void write( Object opt, int recordSize, byte data[] )
    throws Exception
    {
    synchronized ( writeLock )                  /* only one at a time  */
      {
      if ( checkOption( opt, MQe.MQe_Adapter_HEADER    ) ||
           checkOption( opt, MQe.MQe_Adapter_HEADERRSP )  )
        writeBytes( intToByte( recordSize ), 0, 4 );    /* write length*/
      writeBytes( data, 0, recordSize );        /* write the data      */
      if ( checkOption( opt, MQe.MQe_Adapter_FLUSH  ) )
        stream_out.flush( );                    /* make sure it is sent */
      }                                         /*                     */
    }
```

writeBytes は内部メソッドで、ソケットに対してバイト配列 (または部分的な配列) の書き込みを行い、エラーが発生した場合には簡単なエラー・リカバリーを試行します。

```
protected void writeBytes( byte buffer[], int offset, int recordSize )
    throws Exception
    {
    if ( buffer != null )                       /* any data ?          */
      {
      /* break the data up into manageable chuncks                     */
      int i = 0;                                /* Data index          */
      int j = recordSize;                       /* Data length         */
      int MaxSize    = 4096;                    /* small buffer        */
      int RetryValue = 3;                       /* error retry count   */
```

```
    do{                                         /* as long as data     */
      if ( j < MaxSize )                        /* smallbuffer ?       */
        MaxSize = j;
      int Retry = RetryValue + 1;               /* error retry count   */
      do{                                       /* possible retry      */
         try                                    /* catch io errors     */
          {
          stream_out.write( buffer, offset + i, MaxSize );
          Retry = 0;                            /* don't retry         */
          }
        catch ( IOException e )                 /* IO error occured    */
          {
          Retry = Retry - 1;                    /* decrement           */
          if ( Retry == 0 )  throw e;           /* more attempts ?     */
          }
        } while ( Retry != 0 );                 /* more attempts ?     */
      i = i + MaxSize;                          /* update index        */
      j = j - MaxSize;                          /* data left           */
      } while ( j > 0 );                        /* till all data sent  */
    }
  }
```

writeLn メソッドは、 0x0A 文字や 0x0D 文字で終わる文字ストリングをソケットに書き込みます。

opt パラメーターには、次の値を設定することができます。

MQe_Adapter_FLUSH
バッファー内のすべてのデータをフラッシュする

MQe_Adapter_HEADER
すべてのヘッダー・レコードを書き込む

MQe_Adapter_HEADERRSP
すべてのヘッダー応答レコードを書き込む

```
public void writeln( Object opt, String data ) throws Exception
  {
  if ( data == null )                           /* any data ?          */
    data = "";
  write( opt, -1, MQe.asciiToByte( data + "¥r¥n" ) ); /* write data  */
  }
```

これで、(非常に簡単な例ではあるものの、) リスナーやコネクターとして開始された自身の他のコピーと通信を行う tcpip アダプターが完成しました。

簡単なメッセージ・ストア・アダプターの例

この例では、メッセージ・ストア用のインターフェースとして使用されるアダプターを作成します。このアダプターでは、標準の Java 入出力クラスを使用してストア内のファイルを操作します。

この例は、MQSeries Everyplace で提供されるアダプターと置き換えることを意図したものではなく、むしろメッセージ・ストア・アダプターの作成方法を簡単に紹介しているに過ぎません。

MQeAdapter の属性を継承して、新しいクラス・ファイルが構成されます。一部の変数は、ファイル / メッセージの名前やメッセージ・ストアの位置といった、このアダプターのインスタンス情報を保持するように定義されています。

オブジェクトには MQeAdapter のコンストラクターが使用されるため、コンストラクターに付加的なコードを追加する必要はありません。

```
public class MyMsgStoreAdapter extends    MQeAdapter
                               implements FilenameFilter
  {
  protected String  filter   = "";              /* file type filter    */
  protected String  fileName = "";              /* disk file name      */
  protected String  filePath = "";              /* drive and directory */
  protected boolean reading  = false;           /* open'd for reading  */
  protected boolean writing  = false;
```

このアダプターでは FilenameFilter をインプリメントするため、次のメソッドをコード化する必要があります。これは、メッセージ・ストア内で特定のタイプのファイルを選択するために使用される、フィルター操作のメカニズムです。

```
  public boolean accept( File dir, String name )
    {
    return( name.endsWith( filter ) );
    }
```

次に、activate メソッドがコード化されます。これは、ファイル記述子から、すべてのメッセージの保持に使用するディレクトリーの名前を取り出すメソッドです。

このメソッド呼び出しの Object パラメーターには、属性オブジェクトを使用することができます。そのようにすると、これはメッセージ・ストア内のメッセージをエンコード / デコードするために使用される属性となります。

このアダプターでは、次のような Object オプションを使用できます。

- MQe_Adapter_READ
- MQe_Adapter_WRITE
- MQe_Adapter_UPDATE

他のすべてのオプションは無視されます。

```
public void   activate( String  fileDesc,
                        Object  param,
                        Object  options,
                        int     value1,
                        int     value2 ) throws Exception
    {
    super.activate( fileDesc, param, options, lrecl, noRec );
```

```
    filePath    = fileId.substring( fileId.indexOf( ':' ) + 1 );
    String Temp = filePath;                     /* copy the path data  */
    if ( filePath.endsWith( File.separator ) )  /* ending separator ?  */
      Temp      = Temp.substring( 0, Temp.length( ) -
                                  File.separator.length( ) );
    else
      filePath  = filePath + File.separator;    /* add separator       */
    File diskFile = new File( Temp );
    if ( ! diskFile.isDirectory( ) )            /* directory ?         */
      if ( ! diskFile.mkdirs( ) )               /* does mkDirs work ?  */
        throw new MQeException( MQe.Except_NotAllowed,
                                "mkdirs '" + filePath + "' failed" );
    filePath    = diskFile.getAbsolutePath( ) + File.separator;
    this.open( null );
    }
```

close メソッドでは、読み取りや書き込みが許されていません。

```
public void close( Object opt ) throws Exception
    {
    reading  = false;                           /* not open for reading*/
    writing  = false;                           /* not open for writing*/
    }
```

MQe_Adapter_LIST、つまり、ディレクトリー内のフィルターを満たすファイルをリストする要求を処理するためには、 control メソッドをコード化する必要があります。また、MQe_Adapter_FILTER、つまりフィルターを設定してファイルのリスト方法を制御させる要求を処理する場合にも、このメソッドのコード化が必要です。

```
public Object control( Object opt, Object ctrlObj ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_LIST   ) )
      return( new File( filePath ).list( this ) );
    else
    if ( checkOption( opt, MQe.MQe_Adapter_FILTER ) )
      {
      filter = (String) ctrlObj;                /* set the filter      */
      return( null );                           /* nothing to return   */
      }
    else
    return( super.control( opt, ctrlObj ) );    /* try ancestor        */
    }
```

erase メソッドは、メッセージ・ストアからメッセージを除去する際に使用されます。

```
  public void erase( Object opt ) throws Exception
    {
    if ( opt instanceof String )                /* select file ?       */
      {
      String FN = (String) opt;                 /* re-type the option  */
      if ( FN.indexOf( File.separator ) > -1 )  /* directory ?         */
        throw new MQeException( MQe.Except_Syntax, "Not allowed" );
      if ( ! new File( filePath + FN ).delete( ) )
        throw new MQeException( MQe.Except_NotAllowed, "Erase failed" );
```

```
      }
    else
      throw new MQeException( MQe.Except_NotSupported, "Not supported" );
    }
```

open メソッドは、メッセージの読み取りとメッセージの書き込みの両方を許可するブール値を設定します。

```
public void open( Object opt ) throws Exception
    {
    this.close( null );                          /* close any open file */
    fileName = null;                             /* clear the filename  */
    if ( opt instanceof String )                 /* select new file ?   */
      fileName = (String) opt;                   /* retype the name     */
    reading  = checkOption( opt, MQe.MQe_Adapter_READ   ) ||
               checkOption( opt, MQe.MQe_Adapter_UPDATE );
    writing  = checkOption( opt, MQe.MQe_Adapter_WRITE  ) ||
               checkOption( opt, MQe.MQe_Adapter_UPDATE );
    }
```

readObject メソッドは、メッセージ・ストアからメッセージを読み取り、正しいタイプのオブジェクトを再作成します。また、activate 呼び出しに属性が指定されている場合は、データの復号と解凍も行います。これは、特別な機能であり、この機能によって要求は読み取りのパラメーターで指定された突き合わせの基準を満たすファイルを読み取り、最初に出現する突き合わせの基準を満たすメッセージを返します。

```
public Object readObject( Object opt ) throws Exception
    {
    if ( reading )
      {
      if ( opt instanceof MQeFields )
        {
        /* 1. list all files in the directory                          */
        /* 2. read each file in turn and restore as a Fields object    */
        /* 3. try an equality check - if equal then return that object */
        String List[] = new File( filePath ).list( this );
        MQeFields Fields = null;
        for ( int i = 0; i < List.length; i = i + 1 )
          try
            {
            fileName = List[i];                  /* remember the name   */
            open( fileName );                    /* try this file       */
            Fields = (MQeFields) readObject( null );
            if ( Fields.equals( (MQeFields) opt ) )     /* match ?     */
              return( Fields );
            }
          catch ( Exception e )                  /* error occured       */
            {
            }                                    /* ignore error        */
        throw new MQeException( Except_NotFound, "No match" );
        }
      /* read the bytes from disk                                      */
      File diskFile = new File( filePath + fileName );
      byte data[]   = new byte[(int) diskFile.length()];
```

```
      FileInputStream InputFile = new FileInputStream( diskFile );
      InputFile.read( data );                   /* read the file data  */
      InputFile.close( );                       /* finish with file    */
      /* possible Attribute decode of the data                         */
      if ( parameter instanceof MQeAttribute )  /* Attribute encoding ?*/
        data = ((MQeAttribute) parameter).decodeData( null,
                                                      data,
                                                      0,
                                                      data.length );
      MQeFields FieldsObject = MQeFields.reMake( data, null );
      return( FieldsObject );
      }
    else
      throw new MQeException( MQe.Except_NotSupported, "Not supported" );
    }
```

status メソッドは、アダプターに関する状況情報を返します。この例では、フィルターのタイプやファイル名が返されます。

```
public String status( Object  opt  ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_FILTER   ) )
      return( filter   );
    if ( checkOption( opt, MQe.MQe_Adapter_FILENAME ) )
      return( fileName );
    return( super.status( opt ) );
    }
```

writeObject メソッドは、メッセージ・ストアにメッセージを書き込みます。また、activate 呼び出し員属性が指定されている場合は、メッセージ・オブジェクトの圧縮と暗号化も行います。

```
public void writeObject( Object  opt,
                         Object  data ) throws Exception
    {
    if ( writing && (data instanceof MQeFields) )
      {
      byte dump[] = ((MQeFields) data).dump( );         /* dump object */
      /* possible Attribute encode of the data                         */
      if ( parameter instanceof MQeAttribute )
        dump = ((MQeAttribute) parameter).encodeData( null,
                                                      dump,
                                                      0,
                                                      dump.length );
      /* write out the object bytes                                    */
      File diskFile = new File( filePath + fileName );
      FileOutputStream OutputFile = new FileOutputStream( diskFile );
      OutputFile.write( dump );                 /* write the data      */
      OutputFile.getFD().sync( );               /* synchronize disk    */
      OutputFile.close();                       /* finish with file    */
      }
    else
      throw new MQeException( MQe.Except_NotSupported, "Not supported" );
    }
```

これで、(非常に簡単な例ではありますが、) メッセージ・ストアに対してメッセージ・オブジェクトの読み取りおよび書き込みを行うメッセージ・ストア・アダプターが完成しました。

このアダプターは、たとえばデータベースや不揮発性メモリーにメッセージを保管させるなど、多種多様なコード化が可能です。

Websphere Everyplace Suite (WES) 通信アダプター

MQSeries Everyplace は、インターネット・ファイアウォールによる保護を通して HTTP 上でアプリケーションを実行することができる洗練されたセキュリティーを提供します。 Websphere Everyplace 通信アダプターの目的は、MQSeries Everyplace アプリケーションが Websphere Everyplace 認証プロキシーを使用して認証を行うことによって、Websphere Everyplace 認証プロキシーを介したメッセージ・フローを可能にすることです。 図49 は、2 つのアプリケーションが Websphere Everyplace 認証プロキシーを通してインターネット上で通信を行う基本的なシナリオを示しています。

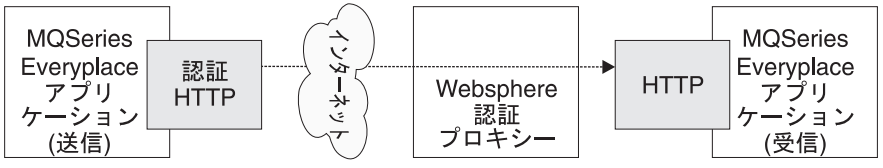

図 49. Websphere 認証プロキシーを通して通信を行うアプリケーション

Websphere Everyplace アダプターは、送信アプリケーション上で認証 HTTP アダプターとして動作します。受信アプリケーションは、同じアダプターを使用することも、MQSeries Everyplace が提供する標準の HTTP アダプターを使用することもできます。

しかし、MQSeries Everyplace の本当の価値は、典型的な同期環境において非同期メッセージングを行えるようにすることです。受信アプリケーションからエンキューされた要求を収集して、時間に関係無くそれらの要求を処理することができます。 303ページの図50 は、入力要求がどのようにして非同期に MQSeries サーバーに到達するのかを示しています。

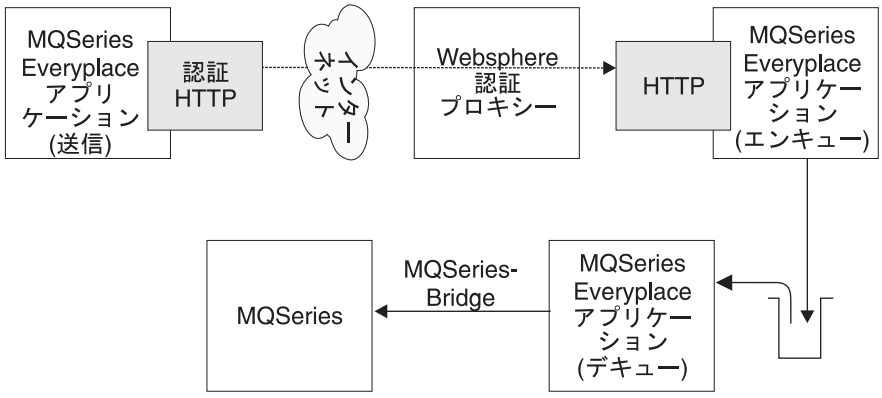

図 50. Websphere 認証プロキシーを通して非同期に通信を行うアプリケーション

これらのそれぞれの環境で、Websphere 認証プロキシーは、受信アプリケーションへのアクセスを制御する機能を追加しています。アダプター・コードは、(アプリケーション提供の) ユーザー ID とパスワード情報を各出力 HTTP 要求に追加することによって、これをサポートします。Websphere 認証プロキシーは、これらの要求を受け入れて、提供された信任状が現在の環境に有効であるかどうか検査します。信任状が有効であれば、プロキシーは要求を受信アプリケーションに転送します。

Websphere Everyplace アダプター・ファイル

標準的な MQSeries Everyplace インストールでは、Websphere Everyplace アダプターは、以下のファイルによって構成およびサポートされています。

...\Java\com\ibm\mqe\adapters\MQeWESAuthenticationAdapter.class
- Websphere Everyplace アダプター・クラス

...\Java\examples\application\Example7.class
- アダプターを使用するコンパイル済みのアプリケーション例

...\Java\examples\application\Example7.java
- アプリケーション例のソース

...\Java\examples\adapters\WESAuthenticationGUIAdapter.class
- Websphere Everyplace アダプターにユーザー・インターフェースを追加するコンパイル済みのアダプター例。他のクラス例と同様に、このクラスは、基本 WES アダプター・クラスに代わるものとしてではなく、ユーザーの要求に合わせて WES アダプターを調整する方法を示すためのものとして提供されています。

...\Java\examples\adapters\WESAuthenticationGUIAdapter.java
- アダプター例のソース

ご使用になっている環境の *CLASSPATH* 変数が MQSeries Everyplace Java フォルダー内のすべてのクラスを検索するように設定されている場合、Websphere Everyplace アダプター・クラス・ファイルは Java 環境内からアクセス可能でなければなりません。ファイルがアクセス可能でない場合には、次のようなコマンドを実行します。

```
set CLASSPATH=%CLASSPATH%;c:\mqe\java
```

これによって、Java が新しいクラスを見ることができるようになります。(このコマンドの正確なフォーマットは、システムによって異なります。)これが完了したら、他の MQSeries Everyplace クラスと同じように Websphere Everyplace アダプター・クラスを使用できるはずです。

Websphere Everyplace アダプターの使用

ここでは、Websphere Everyplace アダプターの使用法について説明します。次の 3 つのパートに分けて説明します。

一般的な操作
ここでは、アプリケーションにおけるアダプターの使用法について詳しく説明します。

認証ダイアログ例の使用
ここでは、クラス例である examples.adapters.WESAuthenticationGUIAdapter の使用法について説明します。このクラスは、基本 WES アダプター・クラスから派生したもので、ユーザーの ID およびパスワードを収集する小規模ユーザー・インターフェースを提供します。

アプリケーション例の使用
ここでは、基本 WES アダプターを使用するように構成されているサンプル・ファイル examples.application.Example7 の使用法について説明します。

このセクションの情報は、Websphere Everyplace 認証プロキシーと MQSeries Everyplace の両方が正しくインストールおよび構成されていることを前提としています。また、MQSeries Everyplace サーバー・キュー・マネージャーおよび MQSeries Everyplace クライアント・キュー・マネージャーが構成されていることも前提としています。

一般的な操作

1. *Network* 別名が com.ibm.mqe.adapters.MQeWESAuthenticationAdapter を指すようにクライアント・キュー・マネージャーの構成 .ini ファイルを変更して、新しいアダプターを使用してメッセージを送信するように、クライアント・キュー・マネージャーを構成します。次のコマンドを使用します。

   ```
   (ascii)Network=com.ibm.mqe.adapters.MQeWESAuthenticationAdapter
   ```

2. 新しいアダプターまたは標準 HTTP アダプターのいずれかを使用して、クライアント・アダプターが提供するデータ・ストリームをデコードするように、サーバー・キュー・マネージャーを構成します。これは、*Network* 別名が com.ibm.mqe.adapters.MQeWESAuthenticationAdapter または

com.ibm.mqe.adapters.MQeTcpipHttpAdapter のいずれかを指すように、サーバー・キュー・マネージャーの構成 .ini ファイル内の行を変更することによって行います。次のいずれかのコマンドを使用します。

```
(ascii)Network=com.ibm.mqe.adapters.MQeWESAuthenticationAdapter
```

```
(ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
```

3. 最初にネットワーク操作を開始する前に、必要なユーザー ID およびパスワードが設定されているように、クライアント・キュー・マネージャー・コードを変更します。たとえば、コードの先頭近くに以下の行を挿入します。

```
com.ibm.mqe.adapters.MQeWESAuthenticationAdapter.setBasicAuthorization("myUserId@myRealm",
                                                                       "myPassword");
```

パラメーターを有効な WES サーバー・ユーザー ID およびパスワードに置き換えてください。

さらに、提供された信任状が無効であった場合には、各ネットワーク操作後に新しい MQeException Except_Authenticate を獲得するように、コードを追加する必要があります。

4. クライアント・キュー・マネージャーが、まだ、プロキシーを介さずにメッセージをサーバー・キュー・マネージャーに送信できるかどうか検査します。
5. プロキシーを通して HTTP 要求を送信するように、クライアント・マシンを構成します。 WES がどのように構成されているかによって、アダプターが*透過プロキシー* または*認証プロキシー* のどちらと動作する必要があるのかが決まります。

透過プロキシー **として構成**

このモードでは、 WES サーバーは、単純な HTTP プロキシーとして動作します。この場合、Java アプリケーションは、プロキシー情報に関連するシステム・プロパティーを設定する必要があります。

http.proxyHost
WES プロキシーのホスト名に設定しなければなりません

http.proxyPort
プロキシーが listen しているポートの名前に設定しなければなりません

http.proxySet
true に設定しなければなりません。これは、透過プロキシー・モードを使用することをアダプターに通知します。

これらのパラメーターは、以下をユーザーの Java アプリケーションに追加することにより、設定できます。

```
  System.getProperties( ).put( "http.proxySet",  "true" );
System.getProperties( ).put( "http.proxyHost", "wes.hursley.ibm.com" );
System.getProperties( ).put( "http.proxyPort", "8082" );
```

クライアント・キュー・マネージャーの宛先 MQSeries Everyplace サーバーへの接続は、 WES プロキシーを使用しない接続に似ています。

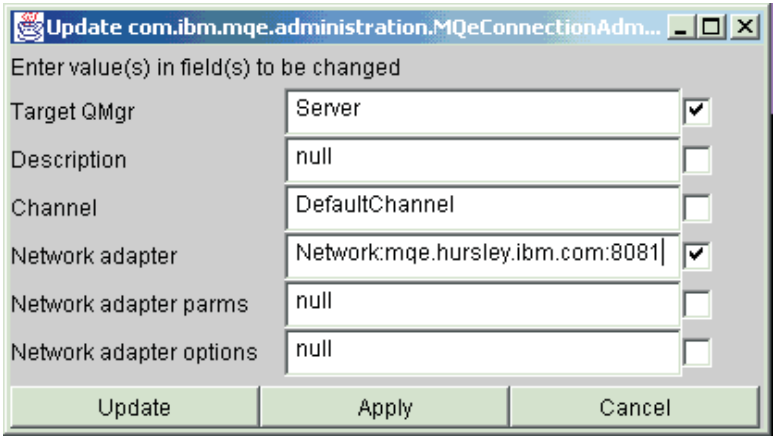

図 51. 管理インターフェース・パネル

新しい設定値を有効にするために、サーバーおよびクライアント・キュー・マネージャーを再始動する必要があります。再始動後、クライアントは、プロキシーを通してサーバーにメッセージを送信できるようになります。

認証プロキシー **として構成**

このモードでは、WES サーバーは、ユーザーが指定する URL に基づいて、要求をサービスに転送します。たとえば、
`http://wes.hursley.ibm.com/mqe` に対する要求を
`mqe.hursley.ibm.com:8082` で稼働している MQSeries Everyplace キュー・マネージャーに転送したいとします。

MQSeries Everyplace からこれをセットアップするには、クライアントのサーバーに対する*接続*リファレンスを更新する必要があります。

宛先ネットワーク・アダプター
認証プロキシー・マシンおよびポートを指していなければなりません

ネットワーク・アダプター・パラメーター
要求されたサービスに対するパス名が含まれていなければなりません

MQSeries Everyplace Example Administration ツールを使用している場合には、「**接続 (Connection)**」を選択してから、「**更新 (Update)**」を選択して、これを構成します。

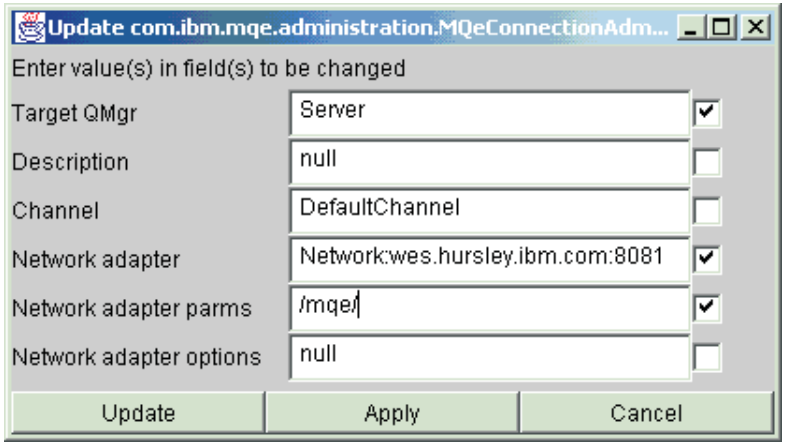

図 52. 管理インターフェース・パネル

注: WES サーバーへのリファレンスは **Network adapter** フィールドに、パス名は **Network adapter parms** フィールドに入力されます。

新しい設定値を有効にするために、サーバーおよびクライアント・キュー・マネージャーをリスタートする必要があります。リスタート後、クライアントは、プロキシーを通してサーバーにメッセージを送信できるようになります。

認証ダイアログ例の使用

ここでは、クラス・ファイルの例である examples.adapters.WESAuthenticationGUIAdapter の使用について説明します。このクラスは、基本 WES アダプター機能に新しい小型ユーザー・インターフェースを追加します。

1. 「一般的な操作」手順のステップ (1) および (2) に従います。ただし、ステップ (1) の 'WESAuthenticationAdapter' を 'WESAuthenticationGUIAdapter' に置き換えます。
2. クライアントの TCP/IP 設定値を、「一般的な操作」のステップ (5) のように構成します。

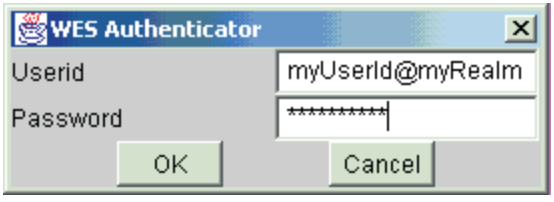

図 53. Websphere Everyplace Suite アダプター・ユーザー・ダイアログ

これで、クライアントは、WESAuthenticationGUIAdapter を使用してサーバーにメッセージを送信できるようになります。このアダプターは、WES アダプターへの書き込み呼び出しを代行受信し、最初の要求で、ユーザー ID およびパスワード情報を要求するダイアログ・ボックスをポップ・アップします。

ユーザーが「**OK**」をクリックするか、**Enter** キーを押すと、「**ユーザー ID (userid)**」および「**パスワード (password)**」フィールドの値を使用して、**setBasicAuthorization()** メソッドが呼び出されます。その後、**write()** が WES アダプターに転送されます。また、ダイアログ・ボックスに「**キャンセル (Cancel)**」ボタンがあり、このボタンを選択すると、要求は WES アダプターに転送されず、現行の書き込み操作が取り消されます。これにより、`MQeException (Except_Stopped)` がスローされます。

認証が失敗すると、次の **write()** 時にダイアログ・ボックスが再表示され、サーバーからの情報が表示されます。認証が失敗したことを認識するために、アダプター例では、**read()** 呼び出しを代行受信して、アダプターからの `Except_Authenticate` MQeExceptions をキャッチします。

注: Web ブラウザーは、通常、最初のフローで認証情報を送信しません。この結果、通常は、レルム情報が含まれている `401` または `407` 応答が返されます。ブラウザーは、この応答を受けてから、認証要求を送信します。ユーザー・クライアントは、この規則に従うことになります。

アプリケーション例の使用

ここでは、アプリケーション・ファイルの例である examples.application. Example7 の使用について説明します。この例は、MQSeries Everyplace のプログラム例である examples.application.Example1 に類似した動きをし、通信に基本 WES アダプターを使用します。

1. 「一般的な操作」手順のステップ (1) および (2) に従います。
2. クライアントの TCP/IP 設定値を、「一般的な操作」のステップ (5) のように構成します。
3. サンプル・ファイル ...\Java\examples\application\Example7.java を編集して有効なユーザー ID とパスワードを挿入し、アプリケーションを再コンパイルします。
4. サーバーを再始動します。
5. 次のコマンドを使用して、Example7 プログラムを実行します。

   ```
   java examples.application.Example7 Server client.ini
   ```

   ここで、

   Server
   リモート・キュー・マネージャー (クライアントが到達方法を認識している) の名前です。

client.ini
クライアントの .ini 構成ファイルを指します。

アプリケーションは、クライアント・キュー・マネージャーを始動し、プロキシーにより認証を行い、メッセージをサーバーに書き込んでから、サーバーからメッセージを取得します。

付録A. MQSeries Everyplace 診断ツール

MQSeries Everyplace には、テクニカル・サポート担当者が問題判別を支援する際に必要な情報を収集するために使用できる簡単な診断ツールが組み込まれています。ツールは、ローカルの MQSeries Everyplace 環境に関する情報を収集します。以下のような情報を収集します。

- CLASSPATH および PATH 情報
- Java システム変数
- MQSeries Everyplace クラスのバージョン情報

このプログラムでは、個人情報や MQSeries Everyplace メッセージ・データは収集されません。 IBM のテクニカル・サポート担当者から要請があった場合にのみ使用してください。

このツールを、実行中の MQSeries Everyplace システムのデバッグ情報を収集するために使用するトレース機能と混同しないでください。

MQeDiagnostics ツールの起動

このツールを使用する必要がある場合には、以下のようにして起動します。

Windows NT/2000 の場合

1. コマンド・プロンプトで、...\mqe\Java\demo\Windows\ フォルダーに移動します。
2. MQeDiagnostics.bat ファイルを、ご使用になっている環境に合わせて編集します。ファイルは、JavaEnv.bat スクリプトを使用するので、必ず、JavaEnv.bat でご使用になっている *CLASSPATH* および *PATH* 環境変数を正しくセットアップするか、または MQeDiagnostics.bat スクリプト内から直接これらの環境変数を構成してください。
3. MQeDiagnostics.bat ファイルを実行して、画面上のプロンプトに従います。
4. ツールが完了したら、MQeDiagnostics.out ファイルを見て、エラーが無いかどうかを調べます。一般的なエラーには、次のようなものがあります。

   .\MQeDiagnostics.properties がありません (".\MQeDiagnostics.properties could not be found")

   このツールでは、入力に MQeDiagnostics.properties ファイルを指定する必要があります。このファイルの正しい位置を指すように、 MQeDiagnostics.bat を編集してから、ツールを再度実行してください。

   com.ibm.mqe.support.MQeDiagnostics が内部または外部コマンドとして認識されません ("com.ibm.mqe.support.MQeDiagnostics is not recognized as an

internal or external command...″)
JavaEnv.bat が正しく構成されていません。必要であれば、MQeDiagnostics.bat および JavaEnv.bat を編集して、ツールを再度実行してください。

″java.lang.NoClassDefFoundError: com/ibm/mqe/support/MQeDiagnostics″
必要であれば、JavaEnv.bat および MQeDiagnostics.bat を編集して、...\MQe\Java\Jars\MQeDiagnostics.jar が *CLASSPATH* 環境変数内で見付かるようにしてください。

注: すべての MQSeries Everyplace クラスがバージョン情報を提供できる訳ではないので、 MQeDiagnostics.out ファイルに「不明なバージョン (Unknown version!)」というメッセージが含まれている場合もあります。

5. MQeDiagnostics.out を MQSeries Everyplace サポート担当者に送信します。

UNIX システムの場合

1. コマンド・プロンプトで、...\mqe\Java\demo\UNIX\ フォルダーに移動します。
2. MQeDiagnostics スクリプトを、ご使用になっている環境に合わせて編集します。ファイルは、 JavaEnv スクリプトを使用するので、必ず、JavaEnv でご使用になっている *CLASSPATH* および *PATH* 環境変数を正しくセットアップするか、または MQeDiagnostics スクリプト内から直接これらの環境変数を構成してください。
3. MQeDiagnostics スクリプトを実行して、画面上のプロンプトに従います。
4. ツールが完了したら、MQeDiagnostics.out ファイルを見て、エラーが無いかどうかを調べます。一般的なエラーには、次のようなものがあります。

.\MQeDiagnostics.properties がありません (″.\MQeDiagnostics.properties could not be found″)
このツールでは、入力に MQeDiagnostics.properties ファイルを指定する必要があります。このファイルの正しい位置を指すように、 MQeDiagnostics.bat を編集してから、ツールを再度実行してください。

com.ibm.mqe.support.MQeDiagnostics : コマンドがありません (″com.ibm.mqe.support.MQeDiagnostics : command not found″)
JavaEnv が正しく構成されていません。必要であれば、 MQeDiagnostics および JavaEnv を編集して、ツールを再度実行してください。

″java.lang.NoClassDefFoundError: com/ibm/mqe/support/MQeDiagnostics″
必要であれば、JavaEnv および MQeDiagnostics を編集して、...\MQe\Java\Jars\MQeDiagnostics.jar ファイルが *CLASSPATH* 環境変数内で見付かるようにしてください。

注: すべての MQSeries Everyplace クラスがバージョン情報を提供できる訳ではないので、 MQeDiagnostics.out ファイルに「不明なバージョン (Unknown version!)」というメッセージが含まれている場合もあります。

5. MQeDiagnostics.out を MQSeries Everyplace サポート担当者に送信します。

その他のシステムの場合

その他のシステムでは、MQeDiagnostics ツールを直接起動します。

1. MQeDiagnostics.jar ファイルをクラスパスに追加します。
2. Java ランタイム環境から com.ibm.mqe.support.MQeDiagnostics クラスを起動します。たとえば、次のように入力します。

   ```
   java com.ibm.mqe.support.MQeDiagnostics MQeDiagnostics.properties > MQeDiagnostics.out
   ```

   プログラムは、MQeDiagnostics.properties ファイルを引き数として使用します。
3. ツールの出力を MQSeries Everyplace サポート担当者に送信します。

付録B. MQSeries Everyplace への保守の適用

保守の更新を適用する場合には、更新に関して出されている指示に従ってください。

保守の更新とその可用性についてのより詳細な一般情報は、
http://www.software.ibm.com/ts/mqseries/ にある、MQSeries ファミリーの Web ページを参照してください。

付録C. 特記事項

本書は米国 IBM 社が提供する製品およびサービスについて作成したものであり、米国以外の国においては本書で述べる製品、サービス、または機能を提供しない場合があります。日本で利用可能な製品、サービス、およびフィーチャーについては、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらのプログラムまたは製品に代えて、IBM の知的所有権を侵害することのない機能的に同等のプログラムまたは製品を使用することができます。ただし、IBM 製以外の製品と組み合わせた場合、その操作の評価と検証については、お客様の責任で行っていただきます。

IBM は、本書で解説されている主題について特許権 (特許出願を含む)、商標権、または著作権を所有している場合があります。本書の提供は、これらの特許権、商標権、および著作権について、本書で明示されている場合を除き、実施権、使用権等を許諾することを意味するものではありません。実施権、使用権等の許諾については、下記の宛先に、書面にてご照会ください。

〒106-0032 東京都港区六本木 3 丁目 2-31
AP事業所
IBM World Trade Asia Corporation
Intellectual Property Law & Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として『現存するままの状態で提供し、』商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

本書に対して、周期的に変更が行われ、これらの変更は、文書の次版に組み込まれます。 IBM は、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire
England
SO21 2JN

本プログラムに関する上記の情報は、適切な条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

著作権:

本書には、さまざまなオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。これらの例は、すべての場合について完全にテストされたものではありません。 IBM はこれらのプログラムの信頼性、可用性、および機能について法律上の瑕疵担保責任を含むいかなる明示または暗示の保証責任も負いません。

商標

次のものは、IBM Corporation の米国およびその他の国における商標です。

AIX
IBM
MQSeries

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

UNIX は、The Open Group がライセンスしている米国およびその他の国における登録商標です。

Windows、および Windows NT は、Microsoft Corporation の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標または登録商標です。

用語集

この用語集は、本書で使用されている用語、および日常用いられている意味合いとは異なる意味合いで使用される語について説明します。場合によっては、1 つの用語にあてはまる定義が 1 つだけではないこともありますが、その定義は本書でその語が使用されるときの意味を表します。

探している用語が見つからない場合は、索引または *IBM Dictionary of Computing* (New York:. McGraw-Hill, 1994) を参照してください。

［ア行］

圧縮機能 (compressor). 転送するデータのサイズが小さくなるよう、メッセージを圧縮するプログラム。

アプリケーション・プログラミング・インターフェース (Application Programming Interface (API)). アプリケーション・プログラミング・インターフェースは、作成するアプリケーションでプログラマーが使用することのできる関数と変数から成り立っている。

暗号機能 (cryptor). 転送中のセキュリティーを提供するため、メッセージを暗号化するプログラム。

インスタンス (instance). インスタンスとはオブジェクトである。クラスがインスタンス化されてオブジェクトが作成されたとき、そのオブジェクトはクラスのインスタンスであるという。

インターネット (Internet). インターネットは、情報を共有する連携公衆ネットワークである。物理的には、インターネットは現在存在する公衆通信ネットワークすべてのリソース全体のサブセットを使用する。技術的に、インターネットは TCP/IP (転送制御プロトコル / インターネット・プロトコル) というプロトコル・セットを使用していることにより、連携公衆ネットワークとして識別される。

インターフェース (interface). インターフェースは、抽象メソッドだけを含みインスタンス変数を含まないクラスである。インターフェースは、メソッドの共通セットを提供するが、そのメソッドは多数の異なるクラスのサブクラスによってインプリメントすることができる。

オーセンティケーター (authenticator). メッセージの送信側と受信側を検査するプログラム。

オブジェクト (object). (1) Java において、オブジェクトはクラスのインスタンスである。クラスはあるもののグループをモデル化し、オブジェクトはそのグループの特定のメンバーをモデル化する。 (2) MQSeries において、オブジェクトはキュー・マネージャー、キュー、またはチャネルである。

［カ行］

カプセル化 (encapsulation). カプセル化は、オブジェクトのデータをプライベートまたは保護にしたり、プログラマーがメソッド呼び出しを通してのみオブジェクトのデータにアクセスおよび操作できるようにしたりするオブジェクト指向プログラミングの技法である。

キュー (queue). キューは MQSeries オブジェクトである。メッセージ・キューイング・アプリケーションは、キューにメッセージを書き込んだり、またキューからメッセージを読み取ることができる。

キュー・マネージャー (queue manager). キュー・マネージャーはアプリケーションにメッセージ・キューイング・サービスを提供するシステム・プログラムである。

クライアント (client). (1)MQSeries Everyplace において、クライアントとは、チャネル・マネージャーまたはチャネル・リスナーを使用せずに稼働する MQSeries Everyplace コードである。サーバー *(server)* の *(1)* と対比。(2)MQSeries において、クライアントは、ローカル・ユーザー・アプリケーションに、サーバー上でキューに入っているサービスへのアクセスを提供する実行時コンポーネントである。

クライアント / サーバー・チャネル (client/server channel). クライアント側からのみ確立できるクライアントとサーバーの間の MQSeries Everyplace の単一方向チャネル。*対等チャネル (peer channel)* と対比。

クラス (class). クラスとは、データおよびそのデータを操作するメソッドのカプセル化コレクションである。クラスのインスタンスであるオブジェクトを作成するために、クラスをインスタンス化することもできる。

携帯情報端末 (personal digital addistant (PDA)). ポケット・サイズのパーソナル・コンピューター。

ゲートウェイ. MQSeries Everyplace ゲートウェイは、MQSeries Everyplace MQSeries-Bridge ・コードを実行するコンピューターである。

［サ行］

サーバー (server). (1) MQSeries Everyplace サーバーは、MQSeries Everyplace チャネル・マネージャーおよび MQSeries Everyplace チャネル・リスナーが構成されている MQSeries Everyplace コードである。これは、複数のデバイスおよびサーバーから同時に受信できる機能を提供する。クライアント *(client)* の *(1)* と対比。(2)MQSeries Everyplace サーバー・コードを実行しているコンピューター。デバイス *(device)* と対比。(3) MQSeries サーバーは、リモート・ワークステーション上で実行するクライアント・アプリケーションにメッセージ・キューイング・サービスを提供するキュー・マネージャーである。(4) より一般的には、サーバーとは、クライアント / サーバーの特定の 2 プログラム間情報フロー・モデルにおいて情報の要求に応答するプログラム、またはサーバー・プログラムを実行するコンピューターである。

サーブレット (servlet). Web サーバー上でのみ実行するように設計された Java プログラム。

サブクラス (subclass). サブクラスとは、別のクラスへ拡張するクラスである。サブクラスは、それ自身のスーパークラスのパブリックおよび保護のメソッドと変数とを継承する。

スーパークラス (superclass). スーパークラスは他の何らかのクラスによって拡張されたクラスである。スーパークラスのパブリックおよび保護のメソッドと変数とは、そのサブクラスで使用可能である。

［タ行］

対等チャネル (peer channel). 通常、クライアント間で使用される双方向の MQSeries Everyplace チャネル。接続は、両側から確立できる。

ダイナミック・チャネル (dynamic channel). クライアントとサーバーを接続して、メッセージの転送を可能にする MQSeries Everyplace チャネルに付けられた名前。これは、要求時に作成されるので、*ダイナミック* という。*クライアント / サーバー (client/server)* および *対等チャネル (peer channels)* を参照。*MQI チャネル (MQI channel)* と対比。

チャネル (channel). *ダイナミック・チャネル (dynamic channel)*、クライアント / サーバー・チャネル (クライアント/サーバー channel)、対等チャネル (peer channel)、および *MQI チャネル (MQI channel)* を参照。

チャネル・マネージャー (channel manager). エンドポイント間の論理多重並行通信パイプをサポートする MQSeries Everyplace オブジェクト。

デバイス (device). MQSeries Everyplace がクライアントとして稼働している小型ポータブル・マシン。サーバー *(server)* の *(1)* と対比。

伝送制御プロトコル / インターネット・プロトコル (Transmission Control Protocol/Internet Protocol (TCP/IP)). ローカル・エリア・ネットワークと広域ネットワークの両方に対して、対等通信接続機能をサポートする通信プロトコルのセット。

同期メッセージング (synchronous messaging). プログラム間で通信するときのメソッドの一つで、このメソッドを使用してプログラムはメッセージをメッセージ・キューに入ることができる。同期メッセージングを使用すると、送信側プログラムはそのメッセージに対する応答を待ってから、自分自身の処理を再開する。*非同期メッセージング (asynchronous messaging)* と対比。

［ハ行］

ハイパーテキスト・マークアップ言語 (Hypertext Markup Language (HTML)). ワールド・ワイド・ウェブ (WWW) に表示する情報を定義するために用いられる言語。

パッケージ (package). Java においてパッケージは、一部の Java コードが特定のクラス・セットにアクセスできるようにする方法である。特定のパッケージの一部である Java コードは、そのパッケージのすべてのクラス、およびそのクラスのプライベートではないすべてのメソッドとフィールドにアクセスする。

パブリック (public). パブリック・クラスまたはインターフェースはどこからでも見ることができる。パブリック・メソッドまたは変数は、そのクラスを見ることができるところからであれば見ることができる。

非同期メッセージング (asynchronous messaging). プログラム間で通信する際のメソッドの一つで、このメソッドを使用してプログラムはメッセージをメッセージ・キューに入れることができる。非同期メッセージングを使用すると、送信側のプログラムは送ったメッセージへの応答を待たずに自分の処理を継続する。*同期メッセージング (synchronous messaging)* と対比。

プライベート (private). プライベート・フィールドはそれ自身のクラス以外からは見ることができない。

ブリッジ (bridge). MQSeries Everyplace と他のメッセージング・システム (MQSeries を含む) との間でメッセージを流せるようにする MQSeries Everyplace オブジェクト。

保護 (protected). 保護フィールドはそれ自身のクラス内、サブクラス内、またはクラスが含まれているパッケージ内からのみ見ることができる。

［マ行］

メソッド (method). メソッドとは、関数またはプロシージャーに対するオブジェクト指向プログラミングの用語である。

メッセージ (message). メッセージ・キューイング・アプリケーションにおいて、メッセージはプログラム間で送信される通信である。

メッセージ・キュー (message queue). 「キュー (queue)」を参照。

メッセージ・キューイング (message queuing). アプリケーションの各プログラムが、メッセージをキューに書き込んで他のプログラムと通信するプログラミング技法。

［ラ行］

ローカル・エリア・ネットワーク (Local area network (LAN)). 限定された地域内でユーザーの構内に配置されたコンピューター・ネットワーク。

［ワ行］

ワールド・ワイド・ウェブ (Web) (World Wide Web (Web)). ワールド・ワイド・ウェブは、プロトコルの共通セットに基づいたインターネット・サービスで、特別に構成されたサーバー・コンピューターが標準的な方法でインターネットを介して文書を配布できるようにする。

J

Java 開発者キット (Java Developers Kit (JDK)). Sun Microsystems 社によって Java 開発者向けに配布されているソフトウェア・パッケージ。その中には、Java インタープリター、Java クラス、および Java 開発ツールが含まれている。Java 開発ツールにはコンパイラー、デバッガー、逆アセンブラー、アプレット・ビューアー、スタブ・ファイル生成プログラム、および文書生成プログラムがある。

Java ネーミングおよびディレクトリー・サービス (Java Naming and Directory Service (JNDI)). Java プログラム言語で指定される API。この API は Java プログラム言語で作成されたアプリケーションにネーミングおよびディレクトリー機能を提供する。

L

Lightweight Directory Access Protocol (LDAP). LDAP はディレクトリー・サービスへのアクセスに用いられるクライアント・サーバー・プロトコルである。

M

MQI チャネル (MQI channel). MQI チャネルは、MQSeries クライアントをサーバー・システムのキュー・マネージャーに接続し、 MQI 呼び出しと応答を両方向に転送する。MQI チャネルは、明示的に作成しなければならない。ダイナミック・チャネル *(dynamic channels)* と対比。

MQSeries. MQSeries は、メッセージ・キューイング・サービスを提供する IBM ライセンス・プログラムのファミリーである。

W

Web. 「ワールド・ワイド・ウェブ (World Wide Web)」を参照。

Web ブラウザー (Web browser). ワールド・ワイド・ウェブ上で配布される情報をフォーマットおよび表示するプログラム。

参照文献

関連資料:

- *MQSeries Everyplace for Multiplatforms Read Me First*, GC88-8656
- *MQSeries Everyplace for Multiplatforms 紹介*, GC88-8653
- *MQSeries Everyplace for Multiplatforms プログラミング・リファレンス*, (SC88-8655)
- *MQSeries An Introduction to Messaging and Queuing*, GC33-0805-01

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

［サ行］

[タ行]

［ナ行］

［ハ行］

［マ行］

［ヤ行］

［ラ行］

［ワ行］

A

B

E

G

J

M

P

R

S

W

Printed in Japan

SC88-8654-02

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12